Analizador de armonía musical

PROYECTO DE SISTEMAS INFORMÁTICOS

Autores:

Víctor Barbero Romero
Carmen Carrión Inglés
Álvaro de los Reyes Guzmán

Departamento de Sistemas Informáticos y Programación
Facultad de Informática
Universidad Complutense de Madrid

Profesor director: Jaime Sánchez Hernández

Curso 2005-2006

Abstract

El análisis armónico es una de las múltiples maneras de conocer cómo una obra musical está hecha. Supone reconocer el tipo de los acordes usados en la obra y, a partir de ello, realizar un análisis más complejo, que nos permite ver cómo esos acordes están relacionados entre sí. Este Analizador de Armonía Musical realiza el análisis armónico de una obra dada, mostrando los resultados mediante una interfaz gráfica. Ha sido desarrollado usando dos tipos diferentes de lenguaje: Java, para la representación gráfica y el tratamiento de la entrada, y Prolog, que ha facilitado la tarea de análisis gracias a su estructura de formalización del conocimiento en hechos y reglas. La entrada a la aplicación se realiza mediante un fichero midi, procesado usando ABC, un sistema de notación musical extremadamente simple, aunque también muy potente. Se usa jMusic, una librería en Java, tanto para reproducir el fichero midi de entrada como para representar la partitura correspondiente. Esta aplicación aprovecha dos lenguajes de diferente propósito para ofrecer al usuario el poder realizar el análisis armónico de una obra dada, dentro de un contexto tonal, definido por las reglas de la armonía que tradicionalmente son consideradas dentro de una tonalidad. El usuario también puede editar la partitura, cambiando por tanto los acordes en la obra, y pedir un nuevo análisis, según las modificaciones realizadas.

Harmonic analysis is one of the multiple ways of knowing how a musical work is made. It means recognizing the kind of chords used in the work and, basing on it, making a deeper analysis, which let us check how those chords link themselves. This Musical Harmony Analyzer performs a harmonic analysis of a given work, showing the results by a graphical interface. It has been developed with two kind of different languages: Java, used for visual representation and for treating the input and Prolog, which has made analysis task easier thanks to his knowledge formalization using facts and rules. The input for the application is took from a midi file. It is processed using ABC, an extremely simple musical notation system, although very powerful too. We use the Java library jMusic as for playing the midi file at input as for showing user the score in it. This application takes advantage of two languages with different purposes in order to bring user a way of make a harmonic analysis of a given work, inside a tonal context, defined by the harmonic rules which traditionally are considered as belonging to a tonality. Users can edit the score too, changing therefore the chords on the work and ask for a refresh of the analysis with the changes made.

Palabras clave

análisis armonía música acordes tonalidad Java Prolog ABC LookAndFeel MIDI

Índice general

Capítulo 1

Introducción

A lo largo de la Historia se han dado múltiples definiciones de lo que es la Música, ya sea de una manera más filosófica, estética o científica. Sin embargo, a partir del siglo XVIII se empieza a establecer, gracias al trabajo de Rameau o de Rousseau que la Música es la unión de ritmo, melodía y armonía (o galantería, según Mattheson). Tomando la armonía como uno de los elementos musicales básicos, tiene sentido llevar a cabo una profundización mayor. A la hora de abordar el análisis de una obra musical, se suelen seguir ciertos pasos que garanticen un recorrido coherente de los elementos y características presentes en la obra, para luego deducir más información de la facilitada por el análisis. En este sentido, todo buen análisis debe empezar por los elementos básicos: atender al ritmo presente en la obra, las características de las melodías y de las líneas utilizadas y también el uso que de la armonía se hace en ella.

Dar una definición de armonía es difícil, ya que tiene múltiples connotaciones. A rasgos generales, la armonía es el arte de combinar sonidos entre sí. En la música occidental, entendemos por armonía una jerarquización de los 12 sonidos posibles, de los que se toman 7 para formar lo que se llama una escala. Desde la Edad Media se fue produciendo un proceso de modalización, en el que se tomaban unos sonidos, a partir de los cuales crear las melodías para usar en una obra. En el siglo XVI se fue asentando una jerarquización de estos sonidos en torno a uno llamado tónica, y más adelante sobre otro denominado dominante. Con el establecimiento de la tonalidad, una selección de 7 sonidos, empezaron a desarrollarse las primeras nociones de lo que hoy distinguimos por armonía tonal en Occidente. Así, surgen las escalas tonales, que están formadas por esos 7 sonidos colocados uno tras otro de manera ascendente o descendente, empezando por uno principal, denominado tónica.

Los acordes son una disposición de notas. También esto es una visión general que, aplicada al campo de la tonalidad, nos lleva a la definición del caso más básico: un acorde tríada está formado por tres sonidos en sucesión, cada uno a intervalo de tercera respecto al anterior, y uno de los cuales da nombre al acorde, denominándose fundamental. En la armonía tonal, sobre cada nota de la escala se forma un acorde, tomando la correspondiente nota como fundamental del mismo. Así se llega al concepto de armonía: en un contexto tonal, donde existe una tonalidad definida que se usa como referente para la composición de una obra, a partir de cada nota del acorde se crean acordes, que toman el nombre dentro de la tonalidad de la nota que usan como fundamental.

En una tonalidad determinada, el acorde formado a partir de la primera nota de la escala se denomina tónica. El formado a partir de la quinta nota de la escala es el llamado acorde de dominante. Los acordes tríada pueden ser de cuatro tipos: perfecto mayor, perfecto menor, de quinta disminuida y de quinta aumentada. Los dos primeros tipos son los más frecuentes en una tonalidad y, de hecho, el llamado modo de la tonalidad viene determinado por el tipo de acorde que se construye a partir de su nota tónica. Así, en la tonalidad Do Mayor, su acorde de tónica es el acorde de Do Mayor. Las tonalidades pueden ser mayores y menores y estar construidas a partir de cualquiera de los 12 sonidos básicos del temperamento occidental.

En una obra que utiliza la armonía tonal, el discurso musical se articula, además de mediante el uso de distintas secciones contrastantes, de la utilización de diverso material melódico o rítmico, mediante los acordes que, tomados de la tonalidad principal de la obra, son presentados a lo largo de la misma. La armonía impregna todo el desarrollo musical, estando presente allí donde haya cualquier nota, ya que ésta cae dentro de la formación de un acorde. Por esto, las obras musicales se presentan como una sucesión de acordes que guían la composición, enriqueciendo el discurso melódico, que es en realidad lo que mayor impresión causa en el oyente. A pesar de esto, la armonía es indispensable en cualquier obra musical que utilice una estética tonal. Aunque sólo nos acordemos de su melodía tras oír una pieza musical, la impresión que ésta nos ha causado está determinada en gran parte por la armonía que acompañaba a la melodía en su desarrollo. No sólo por la armonía, pero también gracias a ella, las obras musicales transmiten un ambiente específico. Esto fue especialmente utilizado a partir del siglo XIX, justo cuando el lenguaje tonal estaba empezando a disolverse.

1.1. Qué es un analizador armónico

Volviendo a una visión más práctica, el analizar una obra musical supone atender al ritmo, a la melodía, a la armonía, a la estructura formal, a los instrumentos utilizados, a la relación de la obra con su contexto histórico y a muchos aspectos adicionales. Sin embargo, de todos ellos quizás es la armonía el que supone un proceso más minucioso: hay que determinar la tonalidad utilizada y a partir de ello, ir comprobando las simultaneidades de notas para deducir a partir de ellas acordes. Cuando estos acordes han sido reconocidos, se debe entonces asignar a cada uno de ellos una función dentro de la tonalidad, que es lo que llamamos el grado del acorde. Puesto que cada tonalidad tiene un acorde formado a partir de las notas de su escala, estos acordes desempeñan un grado dentro de la misma. Los grados se ordenan usando los números romanos, haciendo corresponder al de tónica el I y siendo los acordes siguientes numerados correlativamente.

Además, cada uno de los acordes puede presentarse en distinta inversión: si tomamos un acorde de tres notas, el acorde puede aparecer en tres posiciones, correspondiendo cada una de ellas a la nota del acorde que se sitúa como nota más grave del conjunto. Cuando la nota más grave del acorde es su fundamental se dice que el acorde está en estado fundamental. Cuando esto no ocurre, el acorde se encuentra en inversión. Para indicar la inversión de un acorde se usan cifrados: son indicaciones numéricas que señalan el tipo de acorde (en algunos casos especiales) y cuál es la nota de las que lo forman que está situada como nota más grave. Los tipos de acorde

son muy variados, incluso muchos autores en la Historia de la Música han creado sus propios acordes (como el Acorde Místico de Alexander Scriabin o el Acorde de la Resonancia de Olivier Messiaen). Como aquí se va a trabajar en un contexto tonal, se utilizarán aquellos tipos de acordes que tengan cabida en una tonalidad: perfecto mayor, perfecto menor, de séptima de dominante, de séptima diatónica, de quinta aumentada y de quinta disminuida.

Las obras musicales suelen utilizar una tonalidad como marco de referencia, a partir de la cual tomar los acordes que la forman y usando la jerarquía tonal derivada configurar el discurso musical. Se contemplan algunos casos en los que, para dar mayor variedad y riqueza, un autor puede pasar de una tonalidad a otra, realizándose entonces lo que conocemos por modulación. La modulación es un proceso por el que la tonalidad principal de una obra es sustituida temporalmente por otra, a partir de la cual se toman los acordes con los que guiar el desarrollo musical. Esto supone una reconsideración de la armonía utilizada en esa sección de la obra, que deberá también ser abordada en el análisis.

Hecho este planteamiento teórico como introducción a los contenidos a usar en este proyecto, cabe afrontar su materialización en nuestro proyecto. El análisis armónico realizado por la aplicación que aquí se presenta pretende complementar la información que cualquier interesado en una obra musical puede conseguir por sí mismo, atendiendo a sus características pero, sin embargo, el análisis armónico supone el tener que aplicar ciertas normas que, aun estando formalizadas, supone un trabajo tedioso. Por este motivo, la presente aplicación se presenta como complemento ante lo que alguien puede constatar en el análisis de una obra musical.

Vamos a usar ficheros midi para la entrada de la información musical a nuestra aplicación. Este tipo de ficheros son una recopilación de mensajes que siguen el protocolo M.I.D.I. *(Musical Instruments Digital Interface)*, pensado inicialmente para la conexión entre sí de módulos encargados de la síntesis y manipulación del sonido junto a otros que facilitan a un usuario la interpretación usando esos dispositivos, como puede ser un teclado electrónico. Un fichero midi consiste en una secuencia de mensajes que, bien interpretados, suponen lo que podríamos denominar la partitura electrónica de una obra musical. Los ficheros midi únicamente tienen en su interior mensajes que, a la hora de su reproducción, son interpretados en nuestro ordenador, haciendo uso del banco de sonidos y de las posibilidades de síntesis que facilitan las tarjetas de sonido. Es esta dependencia del banco de sonidos disponible la que provoca que un fichero midi no sea interpretado de la misma forma en dos ordenadores distintos, salvo que usen el mismo banco de sonidos.

El formato digital más común para el almacenamiento musical es el mp3, pero sin embargo, en este tipo de archivo sólo se almacena una compresión de la onda de sonido resultante que, transformada, provoca la reproducción directa de la obra grabada. El formato midi supone una especialización: almacena la información de cada instrumento que interviene en la obra de manera separada por pistas, además de que cada nota está almacenada de manera que es accesible. Simplificando enormemente, se puede considerar al fichero midi como la partitura de la interpretación que luego puede ser recogida en un fichero mp3.

En la aplicación que en este documento se presenta, se toma la información musical almacenada en un fichero midi para, mediante su tratamiento, realizar el análisis armónico de la obra recogida, desarrollándose una interfaz que facilite al usuario el

entender la información conseguida en el análisis, además de su presentación de una manera sencilla e interesante. De esta manera, se cumple el propósito del desarrollo: un Analizador Armónico es una aplicación que, tomando una obra musical como entrada, devuelve información sobre los acordes que en ella se utilizan, dentro de una secuencia armónica y en el contexto de una tonalidad.

1.2. Motivación

El propósito principal de este desarrollo es contribuir mediante el uso de técnicas informáticas a los procesos que se realizan en otros campos del saber, tomando en este caso el de la Música. El análisis armónico, al estar sujeto a normas, puede ser formalizado. Aquí se pretende, tomando ese conjunto de reglas que rigen el análisis, desarrollar una aplicación que, a partir de una obra musical almacenada en un fichero midi como entrada, genere un análisis armónico completo mediante el seguimiento de las normas a aplicar. Dentro del análisis musical, como se ha dicho antes, el análisis armónico es uno de los procesos que conlleva más tiempo y a la vez el que más grado de normas a aplicar tiene. Pero el uso de una aplicación informática facilita enormemente la tarea del análisis armónico, ya que el tiempo de cálculo es inmensamente reducido respecto a lo que tardaría una persona y que la información puede ser presentada de manera visual de tal forma que se produzca una fácil comprensión por parte del usuario.

A la hora de realizar un análisis armónico hay que tener conocimiento sobre las reglas a aplicar en cada momento. En este trabajo hemos seguido el manual de armonía de Walter Piston, que supone un recorrido de las normas de la armonía desde el nivel más bajo, partiendo de la concepción de acorde y de tonalidad, hasta conceptos de desarrollo armónico que no tienen cabida en nuestro propósito. No cabe duda de que, al igual que ocurre en otros ámbitos del conocimiento, este tipo de tareas está sujeto a los conocimientos personales del observador y muy especialmente al nivel de profundidad al que se quiera analizar. Aprovechando la capacidad de cálculo, nuestra aplicación va a realizar un análisis exhaustivo, intentando profundizar al más bajo nivel de detalle, con lo que se garantice que el análisis es lo más detallado posible. Sin embargo, esto puede no convencer a todo tipo de personas que se dediquen al análisis musical. Es comprensible que ante un campo en el que hay que aplicar grandes dosis de intelectualización haya autores que desprecien el análisis armónico, o que por otro lado no le den la importancia suficiente y sólo lo realicen de manera superficial. El desarrollo de la Historia de la Música nos ha confirmado que esto no es suficiente: simplemente mediante la observación de los procesos armónicos que ocurren en una obra ya se puede contextualizar en gran parte dentro de algún período histórico. Además, muchos autores se han caracterizado por el tratamiento personal que han realizado de la armonía y, por si esto fuera poco, ya se ha indicado anteriormente que la armonía es un elemento básico de la Música, sin el cual la experiencia sensorial que nos produce no sería tan completa.

El desarrollo de esta aplicación también tiene por propósito ayudar a completar aquellos trabajos que se encuentran parcialmente elaborados. Por ejemplo, en un fichero midi se almacenan las notas de manera enarmónica: cuenta lo mismo un Do sostenido que un Re bemol, al tratarse del mismo sonido en el sistema temperado igual, pero desde un punto de vista armónico, no podemos aceptar la equivalencia. Y

esto es uno de los grandes retos de las aplicaciones que hacen edición de partituras: cuando tratan un fichero midi, al abrirlo, muestran indistintamente las notas con alteración como sostenidos o bemoles. Es el caso, por ejemplo, de Finale. Al no haber en el fichero midi ninguna indicación de la tonalidad utilizada en la obra, estos programas no pueden resolver qué nombre debe tener cada sonido codificado en el archivo. Nuestra aplicación podría ayudar a esta labor, realizando un análisis en segundo plano que, mediante la determinación de la tonalidad, facilite distinguir si un sonido debe ser representado como una nota con sostenido o la correspondiente con bemol.

Pero las posibles funciones que nuestra aplicación puede desarrollar como ayuda a otros productos no quedan aquí. Tener un módulo que realice un análisis armónico facilita también la generación musical: imaginemos un programa que, dada una obra musical, sepa distinguir la melodía principal y tenga capacidad de generar acompañamientos. El uso del módulo de análisis de nuestro trabajo facilitaría enormemente la tarea de la generación de un acompañamiento que mantuviera la armonía existente en la obra de entrada, pero enriquecida con un acompañamiento distinto. Por ejemplo, en la Facultad de Informática de la Universidad Complutense se desarrolló durante el pasado curso académico Genaro, que podría ser utilizado para, tomando una obra, obtener su secuencia armónica, eliminar el acompañamiento que tiene y generar un nuevo utilizando Genaro.

El diseño modular de la aplicación, que más adelante será indicado, supone tener el motor de análisis en Prolog. Además, este motor también es modular, puesto que el análisis se lleva a cabo en distintas fases, desarrolladas por módulos independientes. Aunque aquí nos hemos centrado en la armonía tonal por la trascendencia que tiene en la música occidental así como ser la armonía con la que más familiarizados estamos, podría modificarse alguno de esos módulos de análisis para aceptar otras armonías, como puede ser la armonía por cuartas de Paul Hindemith. El diseño modular de la aplicación favorece que haya mejoras que se puedan realizar rápidamente, mientras que los demás módulos se mantienen invariables.

Desde un punto de vista más general, el desarrollo de esta aplicación puede suponer el primer paso hacia otras aplicaciones que continúen el análisis. Teniendo la estructura armónica de una obra es más fácil llevar a cabo otros análisis más avanzados, como puede ser el reconocimiento de estructuras formales, de desarrollos melódicos, incluso el reconocimiento temático debido a la repetición de un acompañamiento armónico asociado. Todos estos trabajos pueden verse ayudados de nuestra aplicación, en el sentido de que ya se dispone de un módulo que lleva a cabo el análisis armónico, base de otros posibles.

Como informáticos, este proyecto supone un reto: trabajar en una aplicación multilenguaje, con módulos conectados pero desarrollados en distintos lenguajes (Java y Prolog) y formalizar un área de conocimiento cuyo tratamiento no es habitual en nuestros estudios. Es una de las líneas de trabajo que se puede seguir a la hora de aplicar la Informática hacia otros campos del conocimiento: cómo en base a unas reglas se diseña una aplicación que pretende satisfacer las especificaciones, en este caso determinadas por la armonía tonal.

1.3. Estado del arte

Estudiar el estado del arte supone atender a otras aplicaciones que, si bien no tienen la misma funcionalidad, sí aportan características similares a la que es nuestro propósito. Hay que indicar que es difícil encontrar una aplicación reservada únicamente al análisis armónico, puesto que generalmente las aplicaciones aprovechan este motivo para trabajar en otra dirección o para profundizar en el análisis.

Una de las aplicaciones a destacar dentro del estado del arte es *CLAM Music Annotator*, desarrollado por el Grupo de Investigación en Tecnología Musical de la Universidad Pompeu Fabra de Barcelona. CLAM es el acrónimo de *C++ Library for Audio and Music*, y está compuesto por varios módulos, uno de los cuales se llama Music Annotator, que es una utilidad para visualizar, comprobar y modificar información musical extraída de un archivo de audio. La información musical la recibe en la entrada como fichero de audio, y no utiliza ficheros midi como nosotros. Esto lleva a tener que utilizar Transformadas de Fourier para adivinar las notas que aparecen en la obra musical y, en base a su potencia y frecuencia de aparición, se produce una estimación de los acordes que son usados en una obra musical.

The Melisma Music Analyzer, desarrollado por Daniel Sleator y Davy Temperley, se define como un potente sistema para analizar música y extraer información de ella. El analizador toma un fragmento musical como una lista de eventos (pudiéndose usar entonces un archivo midi) y extrae información sobre el ritmo, el compás, la estructura en frases, la estructura del contrapunto y también la armonía. Supone un trabajo a bastante buen nivel, puesto que resuelve los problemas que nosotros también encontramos, aunque quizás de una manera simplista. Ante el problema de la detección de la tonalidad, divide la obra de entrada en fragmentos en los que se distingue el uso de una tonalidad distinta a la de los demás. Facilita el uso de varios métodos a la hora del este cálculo, como pueden ser estrategias bayesianas, el método de Krumhansl-Schmuckler, o el modelo CBMS. Al igual que nuestro desarrollo, se basa en tomar los distintos resultados que estos métodos han facilitado para, a partir de ellos, deducir la tonalidad principal en cada fragmento. Una vez determinada esta tonalidad, realiza un análisis de los acordes, asignando a cada uno de ellos su grado en la tonalidad principal. Las diferencias con lo que pretendemos en nuestro desarrollo son evidentes: esta aplicación utiliza salida en modo de texto para mostrar sus resultados, en realidad a modo de depuración, puesto que se ofrece también como complemento hacia otras aplicaciones. Nosotros pretendemos dar una visión más continuista de la tonalidad: cómo incluso en los cambios de tonalidad los acordes presentes pueden ser encajados en la tonalidad principal mediante el cálculo de modificaciones. Como contrapartida, este desarrollo ofrece más métodos heurísticos que los que nuestro desarrollo va a utilizar.

La *Harmony Library* es una librería dinámica desarrollada en Power Basic por Godfried-Willem Raes, profesor doctor del Real Conservatorio e Instituto Superior para la Música de Gent (Bélgica). Es un trabajo que lleva dos años sin ser actualizado. Se ofrece como librería adicional para cualquier lenguaje capaz de usar DLL, que incorpora un conjunto de procedimientos y funciones que implementan reglas de la armonía tradicional. Permite una descripción de procesos armónicos, mediante el uso de métodos que facilitan el trabajo con acordes, notas, secuencias de acordes y reglas de armonía. Utiliza una codificación en enteros similar a la que se va a usar en

este desarrollo, mientras que destaca por ofrecer un trabajo complementario a otra aplicación, al mostrarse como librería dinámica. Como contrapartida, no dispone de ningún módulo de visualización de resultados, dejando esta labor a quien vaya a usarla para su aplicación.

En cuanto a los grandes programas de edición de partituras y de sonidos, no se puede destacar ninguna función específica para el análisis armónico dentro de ellos. Rosegarden dispone de un módulo de análisis: si en la ventana de edición de una parte de piano, por ejemplo, se escriben algunas notas y tras seleccionarlas se accede al Harmony Browser, se realiza una estimación de qué acorde pueden formar esas notas, y entonces así aparece indicado. Sin embargo, no ofrece un análisis más exhaustivo, profundizando en un reconocimiento tonal ni en un nivel de detalle como el que pretendemos en nuestra aplicación. Otro editor de partituras, Sibelius, dispone de una opción como la comentada anteriormente: realiza adaptaciones de obras de entrada para agrupaciones instrumentales a petición del usuario, pero la solución no pasa por la armonía: produce un reparto de las voces existentes en los datos de entrada y un pequeño análisis armónico para completar en vertical acordes cuando a alguna de las voces que se está escribiendo no se le puede asignar una nota tomada de la obra original. Este tipo de aplicaciones se verían enriquecidas si incorporaran un módulo de análisis armónico como el que se pretende desarrollar, ya que además se solucionarían los problemas de enarmonía entre notas que se han comentado anteriormente.

Capítulo 2

Detalles de la implementación

A continuación, vamos a describir todos los detalles referentes a la implementación, es decir, las herramientas y técnicas usadas, el motivo por el que hemos escogido dichas herramientas y la arquitectura software de la aplicación.

2.1. Herramientas y recursos utilizados

Para implementar tanto la interfaz como el procesador midi, se ha escogido Java.

Cuando empezamos a plantearnos qué lenguajes íbamos a usar para la implementación, pensamos en C++ para la mayor parte de la aplicación y Prolog para el analizador. C++ ofrecería potencia suficiente y rapidez a la aplicación. Para C++ además, existen útiles librerías para realizar interfaces. Buscamos aplicaciones en C++ de código abierto parecidas a la nuestra como pueden ser los editores de partituras. Pero la verdad es que no encontramos gran cosa, porque la mayoría no eran de código abierto y otras que sí lo eran, como por ejemplo *RoseGarden*, eran demasiado complejas, lo que dificulta enormemente la posibilidad de modificar el código fuente.

Después de hacernos a la idea de usar C++, encontramos una librería, llamada JMusic, que, aparentemente, nos iba a ayudar muchísimo en todo lo referente a representar y editar partituras, abrir y salvar midis, etc. En definitiva, JMusic nos aportaba bastante más cosas que pudiéramos aprovechar que cualquier otra librería o aplicación que habíamos encontrado. Llegados a este punto, cambiamos de opinión y decidimos implementar en Java, que además, nos aporta más facilidades que C++ para implementar interfaces gráficas gracias a la librería Swing y es un lenguaje con el que estamos mucho más familiarizados y por lo tanto, las dificultades en la implementación iban a ser menores. El problema de Java es el rendimiento, pero, al no tratarse de una aplicación crítica, era algo que se podía asumir. Además, Java nos daba la posibilidad de hacer un sistema multiplataforma, como así ha sido, y gracias a eso, nuestra aplicación funciona tanto en Windows como en Linux y seguramente con muy pocos cambios funcionaría perfectamente en ordenadores Macintosh. En la sección de problemas y soluciones veremos que no todo salió como esperábamos con JMusic.

Debido a estos problemas, implementamos la parte que leía y obtenía datos del fichero MIDI gracias al programa *midi2abc*, que dado un fichero MIDI como parámetro producía un archivo de tipo abc con el contenido del midi codificado en este lenguaje. ABC es un leguaje específicamente diseñado para notar partituras

con formato de texto. La ventaja que hemos tenido al utilizar ABC ha sido el tratar directamente con un archivo de texto y poder convertirlo a nuestro formato a través de un parser.

Para la interfaz, también hemos usado otras librerías, las comunmente conocidas como *Look and Feels*. Estas librerías sirven para cambiar el aspecto que Java nos da por defecto para una aplicación. De hecho, el aspecto `NimRod` con el que comienza la aplicación cuando la ejecutamos, no es el aspecto por defecto que nos da Java, si no que es un *Look And Feel* que hemos usado.

Para hacer más sencilla la implementación, utilizamos el popular entorno de desarrollo Eclipse que nos ofrece una amplia gama de facilidades a la hora de programar. Mientras programamos, nos avisa de los errores de sintaxis y de tipos, nos muestra las advertencias, es capaz de hacer los métodos *get* y *set* autómaticamente para los atributos de una clase, nos genera el Javadoc, puede corregir la identación del código, etc. Es una herramienta muy útil y gratuita, que puede descargarse de `www.eclipse.org`. El único incoveniente que tiene es que necesita un equipo relativamente potente para funcionar bien.

Para la generación de los logotipos e iconos de la aplicación, hemos usado la herramienta *Corel PaintShopPro*, que es un editor gráfico bastante potente y sencillo de usar.

Para la realización de las páginas de ayuda en html de la aplicación, hemos usado el editor NVU de Mozilla, para hacer esta tarea mucho más rápidamente que implementando el código html a mano. NVU puede descargarse de `www.nvu.org`

Para el módulo de análisis de la aplicación se ha escogido Prolog debido a sus características y cómo beneficiarían nuestro trabajo. Como primer punto a destacar, Prolog proporciona trabajar fácilmente en la búsqueda de soluciones mediante la aplicación de reglas, que es uno de los principios fundamentales del análisis. La propia estrategia de resolución de objetivos que sigue el lenguaje hace que, definiendo una base de hechos en la que queden reflejadas las reglas de la armonía, lanzando a ejecución un predicado se busque la solución que satisfaga esas reglas. Esto se ha realizado a múltiples niveles: en el Análisis Nominal, quizás hubiera sido mejor que el cálculo de la forma prime se hiciera mediante algún lenguaje imperativo, ya que son más apropiados para el cálculo numérico. Sin embargo, a la hora de reconocer acordes el trabajo en Prolog facilita enormemente la tarea: basta usar alguna condición en los parámetros de cada definición de un predicado para así ya poder distinguir entre casos distintos y es el propio motor de ejecución el que se encarga de ir aplicando las reglas definidas según el caso que se presente.

La facilidad que Prolog ofrece a la hora de trabajar con listas hace que el proceso de análisis sea más fácil para el desarrollador: la creación de listas se realiza automáticamente en cada definición de un predicado en base a los casos que luego son aplicados por el motor de ejecución. A la hora de intentar reconocer un acorde que no está en una forma pura, la definición de reglas mediante hechos hace que consecutivamente el módulo en Prolog las vaya intentando aplicar hasta tener éxito. Esto permite definir reglas de manera más clara, mediante distintos predicados, y que a la hora de interpretar el código fuente sea más fácil su comprensión. El reconocimiento de acordes realizado en el Análisis Musical también se ve favorecido por el uso de Prolog en el caso de querer contemplar más tipos de acordes para su reconocimiento: basta con añadir una definición del predicado *reconAcor* tal que

encaje con la forma prime correspondiente al nuevo tipo de acorde y a partir de ello, definir las normas de tratamiento.

El uso que en basico.pl se hace de reglas básicas de la armonía supone una facilidad a la hora de extender el módulo hacia otro tipo de armonías: cambiando las definiciones realizadas en este archivo se puede conseguir un reconocimiento de otros lenguajes, y también mediante la correspondiente modificación en analisis.pl se puede disponer de un Análisis Nominal que reconozca cualquier tipo de armonía que pueda ser formalizada. Este tipo de definición de reglas aprovecha la sintaxis de Prolog para presentarse de una manera más clara, y con los distintos tipos de acorde separados en definiciones del predicado. En un lenguaje imperativo este proceso de reconocimiento de acordes hubiera supuesto grandes estructuras `if...then` para comprobar cada caso encontrado con un acorde, además de no facilitar tanto la modificación de las reglas formalizadas o la inclusión de otras nuevas.

Para el reconocimiento de la tonalidad, en Prolog nos vemos beneficiados gracias a la posibilidad de buscar todas las soluciones mediante el predicado *findall*. En la heurística que contaba para todas las tonalidades cuántos acordes caen bajo cada una de ellas se utiliza este tipo de técnicas para obtener una lista con los resultados y luego, aprovechando la facilidad con que Prolog maneja listas, escoger la mejor solución. Quizás aquí no hay mucha mejora respecto a lo que se podría haber obtenido usando un lenguaje imperativo, pero también es cierto que este lenguaje declarativo facilita mecanismos para ciertas operaciones de proceso más secuencial. En el Análisis Sintáctico también hay una mejora sustancial gracias al uso de Prolog. En primer lugar, la intención de maximizar la zona de modulación, buscando todas las tonalidades posibles se ve de nuevo favorecida gracias a la búsqueda de todas las soluciones que facilita Prolog. Aquí se maneja una gran cantidad de información para cada posibilidad, que sin embargo no supone demasiado problema gracias al tratamiento en memoria que realiza Prolog con las listas. En un lenguaje imperativo el consumo de memoria hubiera sido superior, salvo que hubiéramos utilizado técnicas de optimización de recursos. De nuevo, las reglas aplicadas, que están definidas en predicados, son fácilmente modificables mediante la nueva definición de los predicados afectados: podría cambiarse la consideración de la búsqueda de una nueva modulación según otro criterio rápidamente, cambiando unas líneas de código sin que el resto del módulo de Análisis Sintáctico se vea afectado.

El propio tratamiento de los acordes dentro de un contexto tonal se puede modificar si, como se ha propuesto antes, se pretende reconocer otro estilo armónico. Aquí hay que redefinir el predicado *sacaGrados*. E incluso, dentro de las definiciones actuales, se puede eliminar el reconocimiento de modulaciones, si se quiere hacer un análisis más plano en el contexto tonal. Es en este módulo donde se pueden ampliar las reglas disponibles para reconocer otro tipo de acordes incluidos en el Análisis Nominal. Supongamos que se añaden las sextas alemanas para su reconocimiento. Esto supondría añadir una nueva regla al Análisis Nominal que reconozca la forma prime correspondiente y dentro del módulo de Análisis Sintáctico una regla que indique cómo tratar ese acorde dentro de una modulación, o si se acepta como modificación de un acorde de la tonalidad.

La manera en que las notas son codificadas mediante enteros, pero almacenada con hechos en salida.pl hace que sea muy rápido modificar la codificación. Cambiando el entero asociado con cada nota se dispone de una nueva posibilidad. Lo mismo

ocurre con todo tipo de asignaciones que se haya hecho a los enteros que internamente se manejan en el módulo en Prolog. El principal beneficio en el uso de este lenguaje está en que al estar basado en hechos podemos cambiar éstos rápidamente y, al derivarse las reglas aplicadas en la ejecución de los hechos indicados, cambia completamente el resultado.

El trabajo del análisis armónico supone tener que manejar muchas posibilidades ante un mismo suceso y escoger entre ellas la mejor, debido a la gran indeterminación que se introduce al considerar las notas como enteros, lo que conlleva que haya que resolver los nombres de las notas por las enarmonías. La principal ventaja que hemos aprovechado con el uso de Prolog es el uso de un lenguaje que permite formalizar rapidamente reglas de muchos ámbitos del conocimiento. Al ser un lenguaje de más alto nivel que otros imperativos que podríamos haber usado, esto nos ha facilitado no atender tanto a cómo se produce la ejecución desde un punto de vista interno y sí poder dedicar más tiempo a la definición de hechos y reglas de derivación con las que se pueda formalizar el análisis armónico. Esto puede verse como una falta de preocupación por el rendimiento, pero sin embargo, Prolog es extremadamente eficiente en el uso de listas y la optimización de memoria que produce cuando éstas tienen una longitud considerable, así como la formalización de reglas de análisis armónico es mucho más fácil y rápida usando un lenguaje declarativo como Prolog que no su materialización en instrucciones imperativas como hubiera sido el caso de usar Java.

Para la edición del código y el trabajo con Prolog se ha utilizado el programa SWI Prolog Editor, desarrollado por Gerhard Röhner, que facilita un entorno sencillo con el que llevar a cabo el proceso de programación. En concreto, tiene realce de la sintaxis del lenguaje, remarcando aquellas palabras reservadas y el entorno de ejecución de SWI Prolog está integrado en el mismo programa, lo que facilita enormemente el trabajo de comprobación y depuración.

Para comunicar la parte en Prolog con la desarrollada en Java podría haberse utilizado los conectores que la implementación de SWI dispone. Esto supondría un trabajo adicional al que hemos seguido nosotros: sabiendo que vamos a usar intercambio de información musical codificada en enteros, y que se va a utilizar un formato reconocible por Prolog para su entrada y una lista de tuplas de enteros para la salida, se puede hacer directamente la comunicación mediante ficheros de texto. En el archivo entradaProlog.txt se escribe una lista con la información pertinente de tal forma que pueda ser leída directamente desde la parte de Prolog como tal, y ser tratada. Para la salida, en el archivo salidaProlog.txt, se dispone de una serie de tuplas de enteros dispuestas en líneas independientes. Usar los conectores hubiera supuesto más trabajo: utilizar las clases y métodos facilitados para después realizar la interpretación de la información almacenada. Aprovechando los procedimientos de lectura de archivos en disco desde Java, se produce una lectura secuencia de líneas de texto, a partir de las cuales se realiza la interpretación de los enteros del archivo. Esto supone un proceso mucho más rápido tanto en el desarrollo como en la ejecución.

Por último, aunque no por ello menos importante, tenemos que destacar el uso de algunos editores de partituras como son *Finale* y *GuitarPro*, que además de proporcionarnos una edición sencilla (aunque tediosa en algunos casos) nos ha permitido exportar la partitura a formato MIDI y modificar los instrumentos asignados

a las pistas MIDI. Gracias a esto hemos podido comprobar que nuestra aplicación funcionaba correctamente.

2.2. Arquitectura de la aplicación

Podemos decir, que la aplicación está formada por tres módulos claramente diferenciados.
El analizador armónico, que es la parte encargada de realizar el análisis a partir de un fichero de entrada llamado entradaProlog.txt. El segundo módulo es el procesador midi que tiene como entrada un fichero midi, se encarga de extraer toda la información necesaria del fichero para el análisis y la representación de la partitura, y genera como salida salidaProlog.txt. Por último, la interfaz, que lleva el control de la aplicación, cuando el usuario quiere abrir un midi, obtiene la ruta del fichero elegido por el usuario, llama al procesador midi y, una vez generado entradaProlog.txt, la interfaz llama al analizador para que haga su trabajo. Por último, la interfaz lee la partitura generada por el procesador midi y el fichero salidaProlog.txt generado por el analizador, y muestra por pantalla el resultado del análisis y la partitura correspondientes.
A continuación describiremos con detalle cada uno de estos módulos.

2.2.1. Analizador armónico

La parte en Prolog del analizador armónico está representada por un ejecutable, hecho con la implementación de SWI de este lenguaje. Recibe la información necesaria para el análisis mediante un fichero de texto, entradaProlog.txt, que la parte en Java deja en el directorio de trabajo. La especificación de este archivo ha sido indicada anteriormente: se trata una lista de tuplas, la primera de las cuales indica el resultado que el Algoritmo de Evaluación de Alteraciones ha encontrado para determinar la tonalidad. El resto de los elementos de la lista son tuplas con dos componentes: un entero, que indica la duración en valores del acorde, y una lista con las notas del acorde, colocadas desde lo más grave a lo más agudo. Para devolver la información del análisis, la parte en Prolog utiliza otro fichero de texto, salidaProlog.txt, cuya especificación se indicará más adelante.

Como módulo de análisis, puede ser dividido en otros tres módulos especializados, cada uno con una función específica, que serán tratados por separado. Hay un primer módulo que realiza un análisis nominal de los acordes recibidos: los procesa, y les da un nombre, como por ejemplo, Do Mayor. Realizado el análisis nominal de todos los acordes encontrados en la lista de entrada de datos, se puede proceder al análisis de en qué tonalidad está escrita la obra recibida y, una vez con la tonalidad calculada, se puede abordar el proceso de asignar a cada acorde reconocido en la primera etapa un grado en la tonalidad correspondiente. Además, en esta tercera fase es cuando se produce el análisis de modulaciones, pretendiendo localizar aquellos puntos en los que se ha abandonado la tonalidad principal para pasar a otra.

Hay que aclarar que en un principio, se va a aplicar el término "acorde" a cualquier disposición vertical de notas encontrada como lista de notas en la lista de tuplas del fichero entradaProlog.txt. A lo que se llama aquí acorde puede no coincidir con la visión tradicional de lo que puede ser un acorde tríada, ya que la implementa-

ción de la parte en Java busca coincidencias verticales de notas para su disposición en una lista, y es la parte en Prolog la encargada de determinar acordes dentro de esas listas, ya en una visión más clásica.

Análisis Nominal

El módulo de Análisis Nominal, entrada a la parte en Prolog, se encuentra en su mayor parte en el fichero analisis.pl. Hay un predicado de entrada al módulo en Prolog, *analizar*, que, aun estando en este fichero, realiza todo el proceso de análisis. Este predicado se encarga de ir llamando a los módulos de manera secuencial, e ir pasando de uno a otro los resultados que son necesarios para cada etapa. La primera operación que se realiza es leer de entradaProlog.txt la lista de acordes a analizar, proveniente de la aplicación en Java. Como en esta lista el primer elemento es una tupla con dos enteros, que indican la tonalidad encontrada por una de las heurísticas para su determinación, tendremos que retirar esta información de la lista de entrada, pues nos hará falta más adelante, y continuar con el resto de la lista, que es donde están los acordes a analizar. Una vez que se ha leído la lista de entrada, se inicializa la salida al fichero de depuración, debugProlog.txt, en el que se recoge toda la información sobre el proceso de análisis, para su tratamiento.

El módulo de Análisis Nominal está representado por el predicado *sacaAcordes*, cuyos parámetros son: el acorde anteriormente inicializado (con valor 0 si se trata del primer acorde a analizar), el acorde actual a analizar, pero en la forma en la que es recibido en el fichero de entrada: `(Duración, ListaDeNotas)`. El tercer parámetro es el resto de la lista de acordes a analizar, seguido del número de compás y un parámetro de salida `Z`, por el que el predicado va a devolver el resultado del análisis nominal.

En cuanto al análisis nominal, los primeros desarrollos de esta aplicación realizaban un trabajo más exhaustivo: para cada acorde recibido en la lista se realizaba un proceso de búsqueda en el que, considerando las notas recibidas, se intentaba determinar de qué acorde se trataba atendiendo a qué tipo de acorde "encajaba" mejor con las notas presentadas. Sin embargo, esto se mostró un trabajo demasiado costoso en tiempo y, sobre todo, con un resultado incierto. Fue aquí cuando se descubrió la *Set Theory Primer for Music*, desarrollada a partir de 1998 por Larry Solomon. Se basa en tomar un conjunto de notas para, a través de operaciones matemáticas con su representación numérica, poder determinar de qué tipo de acorde se trata y su fundamental. En el caso de nuestro análisis, podemos saber también en qué inversión se encuentra el acorde, ya que tenemos como primer elemento de la lista de notas del acorde la nota más grave, a partir de la cual determinar la posición del acorde.

La *Set Theory Primer* supone calcular la llamada forma prime de un acorde: una lista de enteros que indica la distancia en semitonos de las notas del acorde respecto a su fundamental. De esta manera, podemos determinar el tipo de acorde de que se trata. El cálculo de la forma prime se realiza en el fichero solomons.pl, a través del predicado *getPrime*. Su primer parámetro es una lista con las notas a partir de las cuales realizar el cálculo, y a continuación tiene dos parámetros de salida: la fundamental encontrada en el acorde, y la lista de enteros que representa la forma prime. El cálculo a través de la teoría de Larry Solomon puede producir un mayor tiempo de cálculo numérico, pero tiene dos ventajas frente al proceso de búsqueda realizado inicialmente: se realiza mucho menos backtracking, puesto que el resultado

en cada fase del cálculo de la forma prime es único, y también el resultado total es único y específico para cada acorde.

Para calcular la forma prime debe cumplirse la especificación que estamos siguiendo en la entrada: las doce notas deben ser numeradas consecutivamente. Da igual a qué nota se asigne el número 0, ya que se utiliza aritmética modular, que permite cambiar el nombre automáticamente sin problemas. Para el cálculo de la forma prime, el primer paso es ordenar de manera ascendente los enteros que representan las notas del acorde a analizar, eliminando a su vez los duplicados. Para esto, se utiliza el predicado *sort*, facilitado por la implementación de Prolog.

A continuación se calcula el *directed-interval vector* mediante el predicado *getDiv*, una lista de enteros que indica la distancia en semitonos de cada nota respecto a la anterior (y de la última a la primera, de manera cíclica). En esta lista hay que localizar entonces el mayor entero, considerando que si hay varios iguales, se toma aquél cuyo número siguiente sea menor. Con este entero determinado, hay que reordenar la lista de notas, poniendo como primera nota aquella que está en la posición siguiente a la que en el *directed-interval vector* se encontraba el mayor entero. La lista de enteros resultantes es la forma normal, calculada mediante el predicado *getNormal*. Por último, para calcular la forma prime, hay que hacer que el primer entero en la lista normal sea cero, restando su valor a los demás, utilizando aritmética modular. De esta forma, se obtiene la forma prime, que determina inequívocamente de qué tipo de acorde se trata. La fundamental del acorde encontrado es precisamente ese primer entero de la lista en forma normal.

El principal problema de determinar el acorde correspondiente mediante el cálculo de la forma prime es la eliminación de duplicados: esto hace que, como se explicará más adelante, en un acorde tríada cuyas notas estén muy repetidas, si se introduce una pequeña nota adicional, al eliminarse todos los duplicados, el acorde quedará como un acorde de cuatro notas, teniendo el añadido la misma importancia que las que tan repetidas estaban en el acorde encontrado.

Volviendo con el predicado *sacaAcordes*, hay que distinguir un primer caso: los silencios en una voz están indicados mediante el valor `-1` en la correspondiente posición de la lista de notas. Por eso, un primer paso a realizar en el predicado *sacaAcordes* es quitar de la lista de notas recibida los enteros *-1* que haya en ella, para proceder luego al cálculo de la forma prime. Si al quitar los enteros *-1* la lista de notas se queda vacía, es que todas las voces estaban en silencio. Por lo tanto, no hay ningún acorde a analizar, y esto se almacena en la lista de resultados del Análisis Nominal mediante una tupla de enteros `(77, Dr)`, donde `Dr` es la duración recibida en la lista de entrada para ese acorde (aunque esté completamente formada por silencios).

El proceso de Análisis Nominal termina cuando el tercer parámetro del predicado, que se refiere al siguiente acorde, es la lista vacía, ante lo que deja de recorrer la lista de acordes mediante el predicado *sacaAcordes*. Para un acorde intermedio de la lista, el proceso a realizar con él es común: se eliminan todos aquellos `-1` en la lista de notas, se calcula la forma prime del acorde recibido, y se ejecuta el predicado *reconAcor*, pasándole como parámetros el acorde anterior, la forma prime calculada, el acorde siguiente, la fundamental encontrada en el cálculo de la forma prime, el acorde actual en la forma recibida (con la indicación de su duración y su lista de notas), el número de compás (para la depuración) y recibiendo una tupla

de cuatro elementos, cuya información será analizada más adelante. En el caso de que el siguiente elemento en la lista no sea un acorde, sino un cambio de compás (representado por el entero 20), se realiza un tratamiento específico, pasando como siguiente acorde el acorde tras el cambio de compás, y realizándose entonces el correspondiente incremento del parámetro que como entero representa el número de compás en el que se realiza el análisis.

El predicado *reconAcor* es el encargado de determinar el acorde recibido, a partir de su forma prime. La distinción entre los distintos casos se realiza precisamente por la lista definida en su segundo parámetro, a partir de la cual ya sabemos de qué tipo de acorde se trata, y podemos trabajar específicamente para él. *reconAcor* genera las tuplas de cuatro elementos que son el resultado del Análisis Nominal. En estas tuplas, el primer entero representa la fundamental del acorde; el segundo entero, su tipo, siguiendo una codificación indicada más adelante. El tercer entero representa el cifrado del acorde, determinado mediante la nota más grave del acorde y su tipo. Por último, el cuarto elemento de la tupla es el entero recibido en la lista inicial, indicando la duración del acorde encontrado.

El predicado *cifrado* está definido en el archivo basico.pl. Tiene cuatro parámetros: por el primero recibe la fundamental del acorde cuyo cifrado se quiere determinar. En el segundo, el entero que indica de qué tipo de acorde se trata, según el cuadro 2.1. En el tercer parámetro, recibe la nota más grave del acorde, a partir de la cual determinar en qué inversión está, y consecuentemente, devolver por el cuarto parámetro el entero que representa el cifrado que debe tener el acorde. Estos cifrados están almacenados en una clase de constantes de la parte en Java, aunque también pueden ser consultados en el fichero salida.pl, donde aparecen asociados a su cadena correspondiente, para ser mostrados en el fichero de depuración. El predicado *cifrado* distingue a partir del tipo de acorde recibido para, según la distancia de la nota más grave a la fundamental, devolver el entero correspondiente al cifrado. En aquellos acordes en los que hay añadidos (como se verá más adelante), y este añadido se encuentra en la voz más grave (lo que, por otro lado, es menos frecuente), *cifrado* recorre la lista en el tercer parámetro: para aquellos acordes cuyo tipo indica que no tiene añadidos (como puede ser el acorde perfecto mayor), en el tercer parámetro hay una lista con un único elemento: la nota más grave del acorde. Sin embargo, aquellos acordes que no están en una forma que podemos llamar "pura" en esa lista se encuentra todo el acorde, para determinar la primera nota que se puede considerar del mismo, y obtener así su cifrado.

Cuando la forma prime calculada del acorde actual nos indica que se trata de un acorde perfecto mayor, perfecto menor o de quinta disminuida (según el cuadro 2.1), el predicado *reconAcor* realiza un proceso básico: obtiene mediante cifrado el cifrado correspondiente, y rellena la tupla del resultado del Análisis Nominal con los enteros que dispone. En el caso de que el acorde se reconozca como séptima de dominante, la forma prime no devuelve directamente la fundamental, sino la tercera del acorde lo que lleva a tener que usar aritmética modular para sumar 8 a la fundamental obtenida y así calcular realmente la fundamental correspondiente.

En el caso de un acorde con séptima diatónica hay que hacer una distinción: supongamos el acorde La Do Mi Sol. ¿Se trata de un acorde de La menor con séptima añadida o de un acorde de Do Mayor con una sexta adicional? Para hacer estas distinciones, *reconAcor* recurre a predicados adicionales: al distinguir su forma

prime, toma el acorde actual y el siguiente y llama al predicado *buscaSeptValida*, definido en basico.pl. A este predicado se le pasa también la nota de la que queremos confirmar si su función es de séptima, obtenida al sumar 3 a la fundamental proporcionada por la forma prime (que, de nuevo, no coincide con la fundamental específica del acorde, sino que se trata de la quinta). El predicado *buscaSeptValida* aplica las normas de resolución de séptimas para saber si la nota indicada es una séptima añadida: recorre el acorde actual y el siguiente, al haber correspondencia entre las posiciones en la lista y las líneas melódicas) hasta encontrar la nota que queremos comprobar. Hecho esto, si la nota siguiente a la presunta séptima está a un tono o semitono descendente, o si se mantiene, se cumplen entonces las normas de resolución de una séptima que se aceptan en la armonía tradicional, pudiéndose considerar entonces una séptima y por tanto, el acorde correspondiente un acorde con séptima diatónica.

En *reconAcor* hay que distinguir entre dos formas prime para el mismo caso: un acorde mayor con séptima diatónica y un acorde menor con séptima diatónica, que dan como resultado una forma prime distinta. En el caso de que *buscaSeptValida* no haya terminado con éxito, la nota propuesta como séptima no desempeña esa función y por tanto se considera al acorde como su correspondiente perfecto menor o mayor con una nota añadida, sin hacer una indicación de esta condición.

Supongamos ahora que el acorde actual es una quinta vacía: únicamente está formado por dos notas (cuantas veces sea repetidas, ya que los duplicados no afectan a la forma prime), a un intervalo de quinta justa. A priori, puesto que estamos determinando únicamente el nombre de los acordes, no sabemos si se trata de un acorde perfecto mayor o menor. Por esto, se define un tipo especial de acorde, indicado como de quinta vacía, cuyo tipo se resuelve en una etapa posterior, cuando ya se tenga información de en qué tonalidad se encuentra la obra, y consecuentemente, se pueda asignar a este acorde el modo que le corresponde en la tonalidad. Se usa aquí un cifrado específico para indicar que la quinta puede encontrarse invertida.

Un problema mayor se encuentra ante el acorde de quinta disminuida: la forma prime sí está determinada, y se devuelve una nota fundamental. Sin embargo, se trata de un acorde cíclico: al haber la misma distancia en semitonos entre las notas del acorde, cualquiera de las tres puede actuar como fundamental, lo que unido a que en ninguna tonalidad haya un acorde de quinta aumentada, complica el análisis. En este caso, se ha adoptado una decisión: delegar la decisión sobre qué acorde se trata a una etapa posterior, en la que se conozca la tonalidad de la obra y a partir de la cual determinar la fundamental del acorde. Por este motivo, la tupla de cuatro elementos resultado del Análisis Nominal se mantiene, pero con los elementos con distinta función: el primer elemento de la tupla es el acorde encontrado, eliminando sus duplicados; el segundo, que indica el tipo de acorde, tiene el entero especial para el acorde de quinta aumentada. El tercer entero, ante la falta de un cifrado a calcular, almacena la nota más grave del acorde, a partir de la cual determinar la inversión una vez conocida la fundamental del acorde. El cuarto entero de la tupla mantiene la duración del acorde recibida.

Se debe contemplar el caso en el que el acorde actual recibido como lista de notas esté formado por una única nota: entonces, el cálculo de la forma prime devolverá una lista con un único entero, correspondiente a la distancia de una nota consigo misma `[0]`. En este caso, ante la imposibilidad de determinar una armonía, se utiliza un

entero especial para el tipo de acorde, que nos permita continuar con el análisis. El cifrado, de igual forma, está indicado por un entero especial, que en la parte en Java está asociado en la clase de constantes con un cifrado vacío.

Una vez repasados los tipos básicos de acordes y cómo son distinguidos mediante la forma prime, cabe abordar el caso de acordes en vertical cuya forma prime no coincida con ninguna de las que indican un tipo de acorde básico. Esto se produce especialmente en aquellos acordes con añadidos (en los que, como se ha visto, la nota añadida pasa a tener igual importancia que los duplicados de las del acorde) o acordes incompletos. Estos casos se contemplan en el archivo analisis.pl tras las definiciones de *reconAcor* destinadas a los casos básicos de acorde. El primero a abordar es aquél caso en el que a un acorde perfecto mayor se le añade una nota ajena. Esto se produce, por ejemplo, si a un acorde de Do Mayor: Do Mi Sol, se le añade un Re o un Fa. En este caso, la forma prime será la correspondiente a un acorde perfecto mayor, pero también reflejará la distancia de la nota adicional respecto a la nota adicional.

Para superar esta situación, al igual que ocurriría en el caso de un perfecto menor, en *reconAcor* se toma la forma prime recibida y se quita de la lista los enteros que corresponden a la forma prime de acorde mayor o menor. En la lista resultante tenemos entonces la distancia en semitonos de la nota adicional respecto a la fundamental devuelta por el cálculo de la forma prime. Sumando esa distancia a la fundamental, obtenemos la propia nota, ante lo que se puede llamar al predicado *buscaNotaAdicional*, definido en el archivo basico.pl. Este predicado recibe la lista de notas de los acordes anterior, actual, y siguiente, así como la nota que queremos comprobar si es adicional al acorde. El primer paso es recorrer las tres listas, descartando sus elementos hasta encontrar en la lista del acorde actual la nota recibida como adicional. Hecho esto, hay que aplicar las normas de la armonía tradicional para comprobar si esa nota es adicional: es una apoyatura si está a tono o semitono ascendente de la nota siguiente; es nota de paso si sigue la línea ascendente o descendente de las correspondientes notas del acorde anterior o posterior sin mostrar un intervalo mayor al tono; es nota pedal si se mantiene entre los tres acordes; es anticipo si coincide con la nota del acorde siguiente y es un floreo si la nota anterior y posterior coinciden y la del acorde actual está a un semitono o tono inferior o superior.

Cuando ninguno de los casos anteriores se cumple, pasamos a considerar que el acorde esté incompleto: puede tratarse de una única tercera. En este caso, si es una tercera mayor, la forma prime devuelta será `[4]`, ante lo que podemos suponer que pertenece al acorde perfecto mayor correspondiente, pero con la quinta del acorde sin aparecer. De la misma forma podemos suponer que se trata de un acorde perfecto menor si la forma prime es `[3]`.

Si ya el acorde no es una tercera únicamente, pasamos a tener mayor consideración: puede tratarse de un acorde incompleto, que por la disposición vertical de acordes que se hace en la parte en Java se ha producido su división en dos acordes cuyo análisis debe ser realizado de una manera conjunta. En este caso, procedemos primero a fusionar con el anterior y luego con el siguiente. En el caso de fusionar con el anterior, tenemos que comprobar que el acorde anterior existe (el primer parámetro es distinto de cero). Tras fusionar la lista de notas del acorde actual con el acorde anterior, restamos todos los `-1` encontrados (pues representan silencios), y

calculamos la forma prime del acorde resultante de la fusión.

Entonces, hay que proceder a una nueva llamada a *reconAcor*, pero esta vez indicando que el acorde anterior no existe (poniendo el primer parámetro a cero), y sustituyendo la duración del acorde siguiente por `-1`, de tal forma que no se produzca una fusión de la fusión actual con el acorde siguiente. El resultado obtenido, en el caso de tener éxito, es devuelto a *sacaAcordes* con la tupla de cuatro elementos obtenida, indicando la duración correspondiente al acorde actual. En el caso de fusionar con el acorde siguiente, hay que comprobar que la duración de éste no sea `-1`, pues así sabemos que no es la fusión anterior. Se sigue el mismo proceso: se fusiona con el siguiente y se vuelve a llamar a *reconAcor* con la lista resultante. Ahora se indica el acorde siguiente con un `0`, para evitar el proceso siguiente.

Si esto tampoco ha resultado efecto, pues *reconAcor* sigue fallando para la fusión de acordes, vamos a probar un caso extremo: considerar, para cada nota del acorde actual, si se trata de una nota extraña al acorde. Para esto, se va a utilizar un predicado definido en el propio archivo, llamado *sustituye*. Este predicado recibe la lista de notas de los acordes anterior, actual y siguiente, así como un entero de control, y devuelve un entero indicando el número de cambios que se ha realizado en el acorde actual con notas del siguiente o del anterior así como el resultado de los cambios. El entero de control facilita saber de qué acorde se están tomando las notas en la sustitución. *sustituye* contempla el caso de dejar la nota tal cual. También va recorriendo las listas, usando *buscaNotaAdicional* para comprobar si la nota actual en el acorde actual cumple las normas de una nota adicional. En este caso, el entero de control facilita saber de qué acorde se toma la nota con la que se sustituye la adicional: si el entero tiene valor `0`, se toma la nota correspondiente en el acorde anterior para sustituir la nota adicional en el acorde actual, mientras que si es `1`, se toma la nota del acorde siguiente. En el caso de que tenga valor `2`, se sustituye indiferentemente.

Esta distinción entre el acorde actual y el siguiente se sigue con un propósito: cuando en *reconAcor* se contempla este caso extremo, primero se comprueba que ni el parámetro correspondiente al acorde anterior o siguiente tenga valor `0`, lo que nos indicaría que o bien se trata del primer acorde de la obra o que el acorde actual es resultado de una fusión de un apartado anterior. En la especificación del archivo entradaProlog.txt se indica que cuando en una tupla de un acorde, el valor de la duración es superior a 20, es que ese acorde se encuentra en una parte fuerte del compás, y que la duración verdadera es el resultado de restar 20 al entero recibido. Que el acorde se encuentre en la parte fuerte del compás nos facilita pensar en la posibilidad de que mayoritariamente sus notas adicionales tendrán efecto sobre el acorde siguiente, lo que nos lleva a pensar que será con las notas correspondientes del acorde siguiente con las que tendrán que ser sustituidas las notas del acorde actual.

En el tratamiento de este caso en el predicado *reconAcor* se utiliza el predicado *findall*, facilitado por la implementación de Prolog, que proporciona una búsqueda completa sobre todos los posibles éxitos de un predicado indicado. En concreto, se genera una lista con los resultados que hayamos indicado. Para el tratamiento del acorde cuando se encuentra en parte fuerte, se realiza un *findall* sobre *sustituye*, de tal manera que se almacena en una lista todas las sustituciones posibles del acorde actual con notas del siguiente, que a través de una llamada a *reconAcor* sabemos

que han producido un éxito en el reconocimiento del acorde. Como tampoco hay que descartar que se sustituyan notas con el acorde anterior, a esta lista se le concatena otra, resultado de realizar un *findall* usando esta vez el entero de control para indicar que se sustituyan notas con el acorde anterior. Finalmente, se realiza otra búsqueda con *sustituye*, esta vez usando el entero de control que permite tomar notas del acorde anterior y del siguiente, concatenando estos resultados a los de la lista anterior.

En el caso de que el acorde se encuentre en parte débil la secuencia de sustituciones es inversa: primero se piensa en sustituir notas tomándolas del acorde anterior; luego del siguiente y por último de los dos. Cuando un acorde se encuentra en parte débil, por lo general podemos asumir que sus notas adicionales están derivadas de las del acorde anterior. Una vez que tenemos la lista con todas las posibles modificaciones del acorde actual tomando notas del anterior y del siguiente, es momento de usar el entero que en *sustituye* indicaba el número de cambios realizados. Para ello se ha definido en el mismo archivo el predicado *daMenorModif*, que recorre la lista con las modificaciones y devuelve la primera aparición cuyo número de cambios sea el menor de la lista. Es aquí cuando tiene sentido el distinguir primero entre si sustituir con el acorde siguiente o el anterior, y por tanto el orden en el que se concatenan las listas de resultados parciales a la general.

Cuando ninguna de estas técnicas ha dado resultado positivo, llegamos al caso en el que no se sabe qué hacer con el acorde actual, y para ello se utiliza una tupla especial en la lista de resultados del Análisis Nominal: `(66, Dr)`, que representa que se ha encontrado un acorde no reconocible con duración `Dr`. Sin embargo, durante las pruebas realizadas en el proceso de desarrollo no se ha encontrado ningún caso en el que se produzca un acorde no reconocido, quizás por la diversidad de posibilidades que se contempla cuando un acorde no está completo o tiene añadidos.

Tipo de acorde	Forma prime	Entero	Cifrados
Perfecto mayor	`[0,4,7]`	`0`	0, 1, 2
Perfecto menor	`[0,3,7]`	`1`	0, 1, 2
Séptima de dominante	`[0,3,6,8]`	`2`	3, 4, 5, 6
Quinta disminuida	`[0,3,6]`	`3`	7, 8, 9
Menor con séptima diatónica	`[0,3,5,8]`	`4`	10, 11, 12, 13
Mayor con séptima diatónica	`[0,1,5,8]`	`5`	10, 11, 12, 13
Quinta vacía	`[5]`	`6`	0, 14
Quinta aumentada	`[0,4,8]`	`7`	–
Nota única	`[]`	8	18

Cuadro 2.1: Correspondencia entre tipos de acorde, su forma prime, el entero con el que se codifican en la tupla de resultado del Análisis Nominal y los enteros usados para la codificación de su cifrado.

Reconocimiento de la tonalidad

Una vez que tenemos determinado el análisis de los acordes recibidos en la entrada, podemos abordar un paso más: intentar esclarecer cuál es la tonalidad de la obra. Debemos recordar llegado este punto que los ficheros midi no tienen indicación de armadura, lo que haría automática la detección de la tonalidad. Además, las notas

son almacenadas de manera en la que no se distingue entre enarmonías: se codifica con el mismo entero un Sol bemol que un Fa sostenido. Por esto, la detección de la tonalidad supone un cálculo adicional. En el predicado *analizar*, tras la llamada a *sacaAcordes*, se produce la llamada al predicado *reconTonal*, definido en el archivo recontonal.pl.

Este predicado es el encargado de, mediante el uso de heurísticas, determinar qué tonalidad se va a considerar como la principal en el análisis. El predicado recibe como primer parámetro la lista de tuplas de cuatro enteros resultado de la fase de Análisis Nominal, además de una tupla con dos enteros con el resultado del Algoritmo de Evaluación de Alteraciones que se ejecutó en la parte en Java al tratar el fichero midi. Su tercer parámetro es de salida, devolviendo una tupla con la tonalidad escogida como principal. La codificación de las tuplas que indican la tonalidad es igual: el primer entero indica de los 12 sonidos cuál es la nota que da nombre a la tonalidad, y el segundo entero indica su modo, tomándose el valor `0` para indicar que el modo es mayor y el `1` para el modo menor.

El predicado *reconTonal* primero coge la lista resultado de la etapa de análisis anterior y ejecuta una llamada al predicado *busqTodasTonalis*, definido en el mismo archivo. Este predicado implementa una de las heurísticas utilizadas para determinar la tonalidad: consiste en, para todas las tonalidades posibles, recorrer la lista de acordes recibida contando cuántos de esos acordes pertenecen a cada tonalidad. Se pretende así reducir las zonas en las que se detecta una modulación maximizando el número de acordes que caen bajo la misma tonalidad. Es una aproximación a la tonalidad de la obra de un modo estadístico, mediante el saber qué tonalidad aparece mayoritariamente en los acordes reconocidos. El predicado de esta heurística recibe la lista de acordes y devuelve la tonalidad que ha obtenido mejor resultado en el conteo.

Para la ejecución de esta heurística lo primero es obtener la lista de todas las tonalidades posibles. Esto se consigue gracias al predicado *daListaTons*. Devuelve una lista con las 24 tonalidades posibles representadas en tuplas de dos enteros con la codificación explicada anteriormente. Aquí se hace un recorrido en dos direcciones del círculo de quintas: partiendo de la tonalidad de Do Mayor, se considera primero el caso de una tonalidad con un sostenido (Sol Mayor), luego el de la tonalidad con bemol (Fa Mayor), luego el de la tonalidad con dos sostenidos (Re Mayor), y así consecutivamente, hasta llegar a cerrar el círculo de quintas. Las tonalidades menores aparecen en la lista inmediatamente después de su relativo mayor, para ser también consideradas. El propósito de recorrer el círculo de quintas a partir de Do Mayor y progresivamente ir aumentando las alteraciones de la tonalidad a considerar está pensado con la idea de que, en el caso de que se produzca igualdad de acordes en una tonalidad, se escoja primero la que menos alteraciones tiene. Esta idea inicial fue mejorada, como más adelante se explica a continuación.

Conseguida la lista de tonalidades, hay que recorrerla asignando a cada tonalidad una puntuación. Esto se consigue mediante el predicado *cuentaAcordTonal*, definido en el mismo archivo. Recibe como parámetros la lista de acordes, la lista de tonalidades, y devuelve una lista de tuplas en la que a cada tonalidad le queda asignado un entero con la puntuación obtenida. Este predicado recorre cada tonalidad en la lista recibida y llama al predicado *cuentaAcordes*, pasándole la tonalidad encontrada en la lista, la lista de acordes y almacenando en su lista de resultados la

puntuación que ese predicado le devuelve para la tonalidad actual.

cuentaAcordes es el predicado que calcula para cada tonalidad la puntuación que obtiene en la heurística. En el proceso de desarrollo se constató que únicamente contando los acordes en cada tonalidad podría haber tonalidades con la misma puntuación, lo cual llevaría a un resultado incierto. Por este motivo, se pensó en afinar el resultado de la heurística mediante un sistema de bonificación: para cada tonalidad a considerar, si se produce una cadencia perfecta (Dominante - Tónica) dentro de ella, entonces se aumenta la puntuación parcial calculada hasta el momento en dos puntos. Aquí se introduce mayor variedad: si la cadencia perfecta se realiza con alguno de los dos acordes como quinta vacía, se pierde esa sensación de pertenencia a la tonalidad, por lo que únicamente se bonifica con un punto adicional. El saber si alguno de los dos acordes es de quinta vacía se puede hacer gracias al segundo entero de la tupla en el resultado del análisis nominal.

También se produce una bonificación en un punto cuando el acorde actual es tónica en la tonalidad considerada. Pero la duda es: ¿cómo podemos saber que un acorde es determinado en una tonalidad? Para esto está definido el predicado *daGradoPropio* en el archivo sintaxis.pl, pues es donde mayoritariamente es utilizado. Este predicado recibe como primer parámetro la fundamental del acorde, como segundo el entero que codifica su tipo y a continuación, en otros dos parámetros, la fundamental de la tonalidad a considerar así como el entero con su tipo. Su quinto parámetro es de salida, para devolver el grado que en la tonalidad recibida representa el acorde recibido, si es que se produce este caso. Su primer paso es reducir el entero que indica el tipo de acorde a una codificación más sencilla, que resume los tipos de acordes posibles en cinco tipos de acordes básicos: perfecto mayor, perfecto menor, quinta disminuida, séptima de dominante y quinta vacía. Esta reducción tiene como propósito facilitar el cálculo posterior del análisis sintáctico respecto a la tonalidad principal. Como no hay ninguna tonalidad con acordes de quinta aumentada, este tipo de acorde no se considera en la simplificación.

Para realizar la reducción, se usa el predicado *reduce*, definido en basico.pl, que recibe como parámetro el tipo del acorde y devuelve el entero simplificado, según el cuadro 2.2. Con esta reducción, el siguiente paso es llamar al predicado *grado*, que permite saber, para una tonalidad, qué grado es un acorde determinado. Recibe como parámetros la fundamental de la tonalidad, el modo y la fundamental del acorde, devolviendo en su cuarto parámetro el grado correspondiente. Aquí no hace falta recibir el tipo de acorde, puesto que efectúa cálculos numéricos a nivel de fundamentales: según la distancia de la fundamental del acorde a la fundamental de la tonalidad, y atendiendo al modo de ésta, se puede saber si ese acorde pertenece o no a la tonalidad y, en su caso, qué grado desempeñaría. En la definición de grado se contempla que en una tonalidad menor haya un acorde a un semitono ascendente de la tónica, facilitando así el reconocimiento del acorde de sexta napolitana, y usando para ello un entero adicional, el `8`, como indicación de grado.

El siguiente paso es llamar a ejecución al predicado *tipoAcorde*, también definido en basico.pl. Este completa el proceso realizado por el predicado explicado anteriormente: recibe el modo de la tonalidad, un grado de la misma, y devuelve el tipo de acorde que ese grado debe tener dentro de la tonalidad recibida. Es aquí donde tiene sentido el haber reducido los tipos de acorde: había dos enteros (`1` y `4`) para representar un acorde perfecto mayor, ya sea con séptima diatónica o no. Pero este

Tipo de acorde	Entero	Reducido	Tipo reducido
Perfecto mayor	0	0	Mayor
Perfecto menor	1	1	Menor
Séptima de dominante	2	3	Séptima de dominante
Quinta disminuida	3	2	Quinta disminuida
Menor con séptima diatónica	4	1	Menor
Mayor con séptima diatónica	5	0	Mayor
Quinta vacía	6	4	Quinta vacía
Quinta aumentada	7	–	–
Nota única	8	8	Nota única

Cuadro 2.2: Reducción de los tipos de acorde a aquellos que pueden ser encontrados en una tonalidad, junto a su codificación en enteros.

añadido no importa a la hora de determinar si ese acorde pertenece a una tonalidad, ante lo que tiene sentido considerar tipos de acorde en su visión simplificada, para no cargar así los casos a contemplar en la definición del predicado *tipoAcorde*. Es en este caso donde se permite la existencia del acorde de sexta napolitana: en una tonalidad menor, el grado 8 (que había sido asociado antes con el acorde cuya fundamental está a un semitono ascendente de la de la tonalidad), debe ser un acorde perfecto mayor. Por último, ante el caso de un acorde de quinta vacía, pertenece a la tonalidad si el grado asociado es de un acorde perfecto mayor o perfecto menor.

Volviendo a *daGradoPropio*, el proceso se resume a: reducir el tipo de acorde recibido a uno de los enteros básicos, comprobar si el acorde recibido pertenece a la tonalidad indicada mediante el predicado *grado* y, a su vez, obtener el grado correspondiente a ese acorde en la tonalidad indicada. Por último, si el modo del acorde coincide con el que ese grado tendría en la tonalidad recibida, podemos afirmar que el acorde pertenece a la tonalidad. Así, en *cuentaAcordes* se permite saber si un acorde pertenece a la tonalidad, y aumentar por lo tanto la puntuación asociada.

Cuando *busqTodasTonalis* ha realizado el cálculo de la puntuación de todas las tonalidades respecto a la lista de acordes recibida, para obtener el resultado de esta heurística de conteo se recurre al predicado *daMasAbundTonal*, definido en el archivo recontonal.pl. Este predicado recibe la lista con las tuplas que asignan a cada dos enteros que representan una tonalidad una puntuación y devuelve aquella tonalidad en la lista con mayor puntuación. En el caso de igualdad de puntuaciones (algo poco probable, debido al sistema de bonificaciones), se escoge la primera tonalidad en la lista cuya puntuación coincida con el máximo, y es aquí donde tiene sentido el haber usado la lista con las tonalidades recorriendo el círculo de quintas, ya que se escogerá la tonalidad con menor número de alteraciones.

La tonalidad resultado de esta heurística es almacenado en una lista temporal, mediante el predicado *ponListaHeurist*, definido en el mismo archivo, que hace un tratamiento especial: recibe el resultado de una heurística a almacenar y la posible lista actual de resultados. Si la tonalidad a almacenar no está en la lista actual, se almacena al final con un contador inicializado a 1. Si la tonalidad a almacenar está en la lista actual, su contador se incrementa en una unidad. Tras almacenar el resultado de la primera heurística, se almacena el resultado de la heurística facilitado por el

Algoritmo de Evaluación de Alteraciones, cuyo valor fue recibido como parámetro en el predicado *reconTonal*. Por último, se realiza una heurística muy simple: tomar el acorde final como determinante de la tonalidad, considerando que lo más frecuente es que una tonalidad termine en tónica.

Almacenados los resultados de las tres heurísticas, se usa el predicado *daMasAbundTonal*, que ya había sido utilizado por una heurística, para obtener, de la lista con los resultados parciales, aquella tonalidad con mayor puntuación, lo que supone que ha aparecido más veces como resultado de una heurística. En el caso de que no haya ninguna tonalidad con puntuación mayor a las demás (es decir, que las tres heurísticas hayan dado una tonalidad distinta), se favorece a la heurística de conteo en todas las tonalidades, tomándose el primer elemento de la lista (su resultado) como tonalidad principal de la obra.

Análisis Sintáctico

Realizado el determinar la tonalidad de la obra con el módulo anterior, hay que abordar el proceso de Análisis Sintáctico de los acordes recibidos. Esto se produce mediante el predicado *sacaGrados*, definido en el fichero sintaxis.pl, junto a todos los auxiliares en esta parte del análisis. Este predicado recibe por un lado la lista de acordes obtenida en el Análisis Nominal, la tonalidad que se ha obtenido mediante el uso de heurísticas y devuelve una lista de tuplas con el resultado del análisis sintáctico. Puesto que la parte en Prolog tiene que devolver a la parte en Java la tonalidad principal calculada, a partir de la cual se ha realizado el análisis, la tupla que recibe *sacaGrados* indicando la tonalidad se pasa como primer elemento de la tupla de resultados. El proceso de Análisis Sintáctico se realiza recorriendo progresivamente la lista de acordes, mediante el predicado *sacaGrados/4*, que recibe como parámetros la lista y la tonalidad a partir de la cual realizar el análisis.

sacaGrados analiza siempre el primer acorde en la lista recibida y devuelve en la lista de resultado una tupla con enteros adicionales que facilitan información sobre el análisis sintáctico. Para continuar, se produce una llamada a *sacaGrados* con el resto de la lista, a partir de la cual se obtiene el resto de la lista de resultados. El caso más sencillo de la definición de este predicado es aquél en el que el acorde actual a analizar pertenece a la tonalidad recibida. Sabemos que el acorde pertenece a la tonalidad usando el predicado *daGradoPropio* que se explicó anteriormente. Cuando esta llamada tiene éxito, el acorde pertenece a la tonalidad, y entonces se produce una llamada al predicado *completa*. Este predicado recibe el tipo de la tonalidad, el grado del acorde y el tipo de acorde. Está pensado para completar en aquellos acordes que eran de quinta vacía su información, indicando el modo que correspondería a ese acorde en la tonalidad de referencia. Cuando el acorde no es de quinta vacía, la llamada a *completa* no surte efecto.

La lista de resultados está formada por tuplas de enteros, que codifican la información correspondiente. Tras el Análisis Sintáctico, las tuplas de resultado ya tienen completa la información del análisis, ante lo que habrá que destinar un entero específicamente a cada dato que se quiera devolver. Como resultado del Análisis Nominal, al módulo de Análisis Sintáctico le llegan tuplas con cuatro enteros. Aquí se amplía esta información con dos enteros más: se mantienen los cuatro enteros recibidos, referentes a la fundamental del acorde, el tipo de acorde reconocido, el cifrado que le corresponde y la duración del acorde. En esta etapa se añaden dos enteros:

uno que indica qué grado es el acorde actual en la tonalidad principal y un entero denominado Modificador, que en aquellos acordes que no pertenecen a la tonalidad principal facilita información sobre qué modificación se ha realizado en el acorde actual respecto a su original en la tonalidad de referencia. En el caso más sencillo de *sacaGrados*, cuando el acorde pertenece a la tonalidad principal, el valor del Modificador es `-1`.

El siguiente caso que se puede presentar es recibir que el acorde actual es de quinta aumentada. Como se ha explicado antes, se ha delegado a esta etapa del análisis la resolución del acorde. Sabemos distinguir la tupla del acorde de quinta aumentada ya que el segundo entero tiene valor 7. El principio que se sigue a la hora de resolver este acorde consiste en buscar aquella posición del acorde que supone tener más notas en la tonalidad principal. Recordemos que un acorde de quinta aumentada es cíclico: cualquiera de sus notas puede actuar como fundamental. Además, el carácter cíclico de este acorde hace que sólo existan cuatro posibles acordes de quinta aumentada. Como se indicó en un apartado anterior, aquí no tiene sentido guardar en la tupla la fundamental del acorde, puesto que todavía no se ha resuelto. Por eso, en la primera posición de la tupla de entrada se encuentra la lista de notas del acorde sin duplicados. Esta lista de notas se va a utilizar para saber cuántas notas del acorde pertenecen a la tonalidad principal.

Para esto se hace una llamada al predicado *daNotasEnTonal*, que recibe por parámetro la lista de notas del acorde, la fundamental y el modo de la tonalidad y devuelve el número de notas del acorde que pertenecen a la tonalidad, así como una lista con esas notas. Para saber si una nota pertenece a una tonalidad sólo hay que comprobar que esa nota sea la fundamental de un acorde que pueda ser un grado en la tonalidad. Por esto tiene sentido la división que se explicó anteriormente: para saber si un acorde pertenecía a una tonalidad, se comprobaba primero si su fundamental pertenecía al acorde, luego qué tipo de acorde correspondería a ese grado y por último si ese tipo de acorde correspondía con el del acorde recibido. Ahora se puede utilizar parte de los predicados de ese cálculo. En concreto, mediante una llamada al predicado *grado* podemos ir recorriendo la lista de notas del acorde, de tal manera que si esa nota pertenece a la tonalidad principal, aumentamos el contador y la incorporamos a la lista del resultado.

Con la lista de notas que del acorde de quinta aumentada pertenecen a la tonalidad, podemos pasar a buscar su fundamental. Para ello, se utiliza el predicado *daFundQuintAum*, que recibe por parámetro la lista de notas del acorde en la tonalidad junto a su número y devuelve la fundamental del acorde. Si pensamos en los cuatro acordes posibles de quinta aumentada, nos damos cuenta que, para una tonalidad cualquiera, tres de ellos tienen dos notas en la tonalidad y el cuarto tiene una. Este hecho se utiliza para determinar la fundamental del acorde, manteniendo que puesto que se trata de un acorde modificado, se busca que haya el mayor número de notas del acorde en la tonalidad. Además, cuando hay dos notas del acorde en la tonalidad, estas dos notas siempre son consecutivas en el acorde, por lo que forman un intervalo de tercera mayor. En este caso, vamos a considerar como fundamental del acorde la nota inferior del intervalo, pues es la que tomándola como fundamental del acorde que tiene a la otra, deja a la tercera nota del acorde de quinta aumentada como modificación de la quinta del acorde correspondiente perfecto mayor en la tonalidad. En el caso de que sólo haya una nota de ese acorde en la tonalidad, esa

misma nota es la que se va a tomar como fundamental del acorde. De esta forma, en el predicado *daFundQuintAum* se procede a calcular si las dos notas recibidas están en intervalo de tercera o sexta (su inversión) o si se trata de una única nota, procediendo entonces a devolver la fundamental siguiendo la norma descrita. Calculada la fundamental del acorde de quinta aumentada, podemos obtener su cifrado, mediante llamada al predicado *cifrado*.

Ahora llega el proceso más complejo del Análisis Sintáctico. Cuando un acorde no pertenece a la tonalidad principal, admitimos que se trata de un acorde modificado, y entonces, para completar la información del análisis, debemos dar datos de cómo ese acorde ha sido modificado respecto al que correspondería en la tonalidad principal. El primer paso a realizar, algo que se entenderá posteriormente, es determinar el ámbito de las alteraciones de la tonalidad: si la tonalidad actual trabaja con bemoles o con sostenidos. Para ello, se utiliza el predicado *daAmbito*, definido en el archivo basico.pl. Este predicado recibe como parámetros la fundamental de la tonalidad y su modo, devolviendo en un entero el ámbito de la misma (`0` para sostenidos y `1` para bemoles). Aquí se produce una primera solución del problema de las enarmonías por la codificación de notas en enteros: es un primer paso hacia distinguir que, por ejemplo, el entero `6` (Fa sostenido o Sol bemol) representa como fundamental a la tonalidad Sol bemol cuando su modo es mayor y no a Fa sostenido, ya que la primera tiene menos alteraciones: 6 bemoles frente a 7 sostenidos. Esta información está introducida a modo de hechos para las tonalidades mayores. En el caso de las menores, su ámbito es el mismo que el de su relativo mayor.

Determinado el ámbito de la tonalidad principal, se produce una llamada al predicado *resuelveGrado*, definido en sintaxis.pl. Este predicado recibe como parámetros la fundamental y el modo de la tonalidad, la fundamental y el tipo del acorde a considerar, así como el ámbito de la tonalidad, y devuelve de qué grado se trata y el entero con el que indicar la modificación que se ha realizado del acorde respecto al de la tonalidad principal. La primera comprobación que se realiza en *resuelveGrado* es ver si la fundamental del acorde corresponde con la de un acorde de la tonalidad recibida, usando para ello el predicado *grado*, que se detalló anteriormente. Si esto se cumple, entonces hay que distinguir dos posibilidades: que el acorde tenga como tipo el que le correspondería siendo acorde de la tonalidad recibida, ante lo que se devuelve `-1` como valor para el modificador. En el caso de que esto no ocurra, hay que devolver un entero para el entero Modificador de la tupla de resultado, con el que indicar qué modificación se ha hecho sobre el acorde de la tonalidad, de tal manera que es un acorde cuya fundamental sí pertenece a la tonalidad, pero cuyo tipo no corresponde con el que debería tener.

Para ello, se utiliza el predicado *cadAjeno*, que hace corresponder el tipo reducido del acorde recibido con un entero que codifica la modificación que se ha realizado del acorde. Atendiendo a los valores que se han indicado en el cuadro 2.2, hay que asignar un entero a la modificación realizada sobre tipos de acordes reducidos representados por los enteros 0, 1, 2, 3 y 7. En esa tabla se había indicado que el entero que representa el acorde de quinta aumentada no tiene reducción puesto que no hay acorde de quinta aumentada en ninguna tonalidad. Sin embargo, en la llamada a *resuelveGrado* desde *sacaGrados* no se había producido una reducción del entero que representa el tipo de acorde, para contemplar este caso como excepción. Los enteros usados como codificación de la modificación de un acorde aparecen asociados a su

Entero	Modificación del acorde
0	Mayor
1	Menor
2	Rebajado
3	Aumentado
4	Rebajado mayor
5	Rebajado menor
6	Aumentado mayor
7	Aumentado menor
8	Quinta disminuida
9	Rebajado quinta disminuida
10	Aumentado quinta disminuida
11	Quinta aumentada

Cuadro 2.3: Valores para el campo Modificador de una tupla de la lista de resultado junto a la modificación que significa en el acorde correspondiente.

significado en el cuadro 2.3.

Volviendo al caso de tratar la quinta aumentada, la llamada a *resuelveGrado* nos ha devuelto el entero a guardar como modificador en la tupla de resultados, con lo que queda resuelto el acorde, determinándose su fundamental, a qué grado pertenece respecto a la tonalidad principal y el entero que debe representar su modificación.

sacaGrados se completa contemplando las situaciones adicionales definidas en la fase de Análisis Nominal: ante un cambio de compás, se mantiene en la lista final de resultado, ocurriendo lo mismo con un silencio general de todas las voces. En el caso de un acorde no reconocido, el propósito es informar al usuario de que no se ha podido realizar ninguna identificación con ese acorde, ante lo que se mantiene en la lista resultado la tupla correspondiente, indicando la duración de ese acorde anómalo.

Ahora podemos plantear otra situación: ¿y si el acorde actual en el Análisis Sintáctico no pertenece a la tonalidad? Podemos suponer dos alternativas: que sea una pequeña modificación de un acorde de la tonalidad, con el fin de dar colorido, o que sea el primer acorde de una zona de modulación. Esta última alternativa es la que primero se comprueba en el análisis. Cuando *sacaGrados* encuentra un acorde que no pertenece a la tonalidad, se realiza una búsqueda de una posible modulación. Para ello, se utiliza el predicado *tonalisPosibles*, definido en el mismo archivo. Este predicado recibe la fundamental y el tipo de acorde y devuelve la fundamental y el modo de una tonalidad a la que puede pertencer el acorde recibido. Para ello, utiliza el predicado *reduce*, para así obtener el entero reducido del tipo de acorde. Luego se produce una llamada a *tipoAcorde* para saber, con ese tipo reducido de acorde y el modo de la tonalidad, qué grado de la misma puede ser el acorde recibido. Por último, se produce una llamada a *sabeGrado*, definido en el archivo basico.pl, que recibiendo la fundamental de un acorde y el grado que desempeña en una tonalidad, devuelve de qué tonalidad se trata, mediante el cálculo con aritmética modular. De esta forma, *tonalisPosibles* es una manera de obtener todas las tonalidades posibles a las que puede pertenecer un acorde.

Y es precisamente ese proceso el que se realiza ante un acorde que no pertenece a

la tonalidad principal: se ejecuta un *findall* sobre *tonalisPosibles*, pasándole el acorde que es ajeno a la tonalidad principal y recibiendo todas las posibles tonalidades a las que ese acorde puede pertenecer. Con esta información, dentro del *findall*, se produce una llamada a *busqAcordOtraTonal*, que recibe el resto de la lista de tuplas proveniente del Análisis Nominal, la fundamental y el modo de la tonalidad que se ha detectado como posible en la modulación y devuelve la longitud de la cadena de acordes que, a partir del acorde actual se ha encontrado en esa nueva tonalidad, el resto de la lista de tuplas de entrada a partir del cual continuar con el análisis y una lista de tuplas de resultado del Análisis Sintáctico.

busqAcordOtraTonal sigue un proceso parecido al de *sacaGrados*, pero con la salvedad de tener que ir contando los acordes que son encontrados en la tonalidad recibida. Ante un acorde en la nueva tonalidad, lo que se comprueba mediante el uso del predicado *daGradoPropio*, se incrementa el contador y se indica en la tupla de resultado la información sobre el acorde. Actúa de la misma manera ante tuplas especiales: los cambios de compás y los silencios generales se mantienen, así como ante un acorde no reconocido, se continúa el análisis. Si en este predicado se llega a un acorde que no pertenece a la tonalidad recibida por parámetro, podemos suponer que entonces la modulación se ha terminado, ante lo que *busqAcordOtraTonal* termina su ejecución, devolviendo el resto de la lista de tuplas a partir del que seguir el análisis y las tuplas de resultado. En estas tuplas de resultado hay que añadir información: además de los enteros para dar información sobre el acorde, hay que indicar qué grado supone el acorde actual en la nueva tonalidad, incorporándose para ello un nuevo entero a la tupla de resultado, que indica este valor.

En *sacaGrados*, al realizarse un *findall* sobre todas las tonalidades posibles y con una llamada a *busqAcordOtraTonal* para cada una de ellas, se dispone en una lista de todos los resultados posibles, junto al análisis en otra tonalidad y el resto de la lista de entrada a partir de la cual continuar el análisis. Aquí se sigue el principio de maximizar la zona de modulación: de todas las cadenas posibles de acordes reconocidos en otras tonalidades, se escoge aquella cadena cuya longitud sea mayor, ya que supone que se ha encontrado un mayor número de acordes en otra tonalidad. Para ello, se utiliza el predicado *daMejorCadena*, que tiene éxito cuando se encuentra una cadena de acordes en otra tonalidad de longitud mayor que `1`. Cuando esto ocurre, se concatena a la lista de resultado que *sacaGrados* estaba generando la lista obtenida en el findall junto a la secuencia de acordes y se continúa el análisis a partir del resto de la lista de entrada que esa secuencia había devuelto.

La modulación se indica en la lista de resultado mediante el uso de una tupla especial de tres enteros: el primero es el entero `21`, que actúa como indicador de la modulación, pues en la parte en Java el tipo de tuplas del archivo de salida se distingue a partir del valor del primer entero de cada una de ellas. Tras este elemento, siguen dos enteros indicando la fundamental y el modo de la tonalidad a la que se va en la modulación. Esta tupla es añadida a la lista de resultado en *sacaGrados*, cuando se ha encontrado una cadena de acordes en otra tonalidad de longitud suficiente. Para indicar que la modulación se ha acabado, se introduce en la lista de resultado el entero `22`, que es devuelto por *busqAcordOtraTonal* cuando se encuentra un acorde que no pertenece a la nueva tonalidad.

Si en esta búsqueda no se encuentra una cadena de acordes de longitud suficiente,

podemos suponer entonces que el acorde actual es una modificación fortuita de un acorde de la tonalidad principal. Para resolver esta situación se produce una llamada a *resuelveGrado*, aportando la información del ámbito de la tonalidad y el tipo reducido del acorde, como se explicó antes. De esta llamada se obtiene el entero que indica la modificación que se ha realizado sobre el acorde de la tonalidad principal, para que sea guardado en la tupla correspondiente.

Cuando se ha realizado todo el Análisis Sintáctico de la lista de entrada, la llamada principal de este módulo, en el predicado *sacaGrados*, lleva a la ejecución del predicado *resuelveNombres*, que recibe por parámetro la lista de resultado de esta etapa del análisis, el ámbito de la tonalidad principal, así como su fundamental y su modo, y devuelve una lista que ya es considerada resultado total del análisis. Este predicado tiene por principal función resolver los casos de enarmonía y calcular la modificación que respecto a una tonalidad se realiza de un acorde en una zona de modulación. Supone recorrer elemento a elemento de la lista que el predicado *sacaGrados* ha generado.

El primer paso para la resolución de enarmonías es considerar el ámbito de la tonalidad. Las notas que producen un problema de enarmonía son aquellas que pueden ser vistas como una nota con bemol u otra con sostenido, como por ejemplo Fa sostenido y Sol bemol. Estas notas están marcadas en el archivo basico.pl, mediante hechos con el predicado *enarm*. El primer paso para la resolución de enarmonías es comprobar si la tonalidad principal es del ámbito de bemoles y si la fundamental del acorde analizado en la tupla actual es una nota que supone problemas de enarmonía. Si este caso se cumple, pues que el nombre de las notas numeradas del 0 al 11 está asignado usando sostenidos, se ha utilizado una numeración adicional, del 22 al 33, que para aquellas notas que pueden ser vistas como enarmonía, proporcionan su nombre en bemol. Por esto, al darse la situación anterior, se suma 22 al entero que indica la fundamental del acorde actual.

Si se llega a una tupla que indica una modulación, se obtiene el ámbito de la nueva tonalidad detectada y se produce una llamada al predicado *resuelveNombresDoble*, que recibe la lista resultado del análisis con la tupla que indica la modulación al comienzo, así como el ámbito de la tonalidad principal y su fundamental y modo, y el de la nueva tonalidad. Al resolver el nombre de la tonalidad principal se llega a una misma situación: puede tratarse de una tonalidad cuya fundamental plantee un problema de enarmonías, ante lo que se realiza la el mismo proceso de comprobación mediante el predicado *enarm* y, en su caso, sumar 22 al entero de la fundamental de la tonalidad para obtener su nombre enarmónico en bemol.

Dentro de *resuelveNombresDoble* ocurre lo mismo con las fundamentales de los acordes: si es un acorde cuya fundamental plantea problemas de enarmonía dentro de una tonalidad del ámbito de los bemoles, habrá que sumar 22 al entero que indica la fundamental para así pasar a indicar que se trata de una nota bemolizada. Además, ya se produzca problema de enarmonía o no, hay que terminar de calcular la información del análisis: si en una secuencia de acordes en la que no hay modulación se proporciona, además de la información del análisis nominal del acorde, qué grado supone en la tonalidad principal, en una zona de modulación hay que complementar esta información con la de qué grado supone ese acorde también en la nueva tonalidad y la posible modificación que se ha realizado del acorde respecto a cómo debería presentarse en la tonalidad principal. Tanto en *resuelveNombres*

como en *resuelveNombresDoble* hay que continuar en el caso de que se encuentre un elemento especial en la tupla, como puede ser un acorde no resuelto, un cambio de compás o un silencio general.

En *resuelveGrado* ya se trató anteriormente la situación en la que el acorde recibido pertenece a la tonalidad indicada, mediante el uso del predicado *grado*. Sin embargo, puede producirse una llamada a este predicado pasando una tonalidad y un acorde cuya fundamental no pertenezca a aquélla, tratándose entonces de un acorde ascendido o rebajado. Para resolver esta situación se atiende a la resolución de enarmonías que se había producido en el predicado *resuelveNombres*: ante un acorde que no pertenece a la tonalidad, como los grados se disponen en éstas siguiendo una distribución de tonos y semitonos, hay que suponer que la fundamental del acorde se encuentra a un semitono de dos grados de la escala de la tonalidad. Para ello, se comprueba si la fundamental del acorde actual es una nota bemolizada: de ser este caso, ese acorde supone una modificación del correspondiente acorde cuya fundamental está a un semitono ascendente de la del acorde recibido. Por otro lado, si la nota fundamental del acorde no es un bemol, vamos a tomar como fundamental del acorde aquella que está a un semitono descendente del acorde. Esto supone una simplificación del problema por el que no podemos saber qué modificación se ha hecho de un acorde en una tonalidad, ya que las enarmonías impiden saber el sentido (ascendente o descendente) de la modificación. Esta decisión que se ha adoptado sobre el cambio que se ha producido en la fundamental del acorde se almacena en una variable cuyo valor es necesario más adelante.

Tras decidir la fundamental que correspondería en la tonalidad a ese acorde si no se hubiera modificado, se produce una llamada a *resuelveGrado*, pasando la nueva fundamental pero el tipo de acorde que se ha encontrado, para así obtener la modificación que sobre el acorde se ha realizado respecto a su posición original en la tonalidad. Esta llamada nos devuelve la modificación realizada sobre el tipo del acorde, según los valores almacenados en el predicado *cadAjeno*, que se explicaron anteriormente. Con esta llamada ahora se puede completar la información sobre la modificación realizada al acorde: sabemos qué modificación se ha realizado del acorde respecto al tipo que debería tener en la tonalidad, y es momento de indicar también el sentido de la modificación, si ha sido ascendente o descendente.

Para ello, se utiliza el predicado *cambiaCA*, definido en el mismo archivo. Recibe como primer parámetro el entero devuelto por la última llamada a *resuelveGrado*, indicando la modificación sobre el tipo de acorde, y también el valor de la variable en la que se había almacenado el sentido de la modificación del acorde (si había sido ascendente o descendente). Usando una estructura de tabla de correspondencia, se encuentra para los valores de los dos primeros parámetros de entrada un valor que corresponde con el entero que codifica la modificación realiza sobre un acorde usando el significado indicado en el cuadro 2.3. De esta manera, se completa cada tupla que proporciona la información sobre el análisis de un acorde con, además de los datos del acorde en sí, independientes de la tonalidad (como es la fundamental, el tipo, el cifrado y la duración), con qué grado supone ese acorde respecto a la tonalidad principal, respecto a la nueva tonalidad y, en su caso, qué modificación se ha realizado en el acorde respecto a la forma en que se debería presentar en la tonalidad principal.

Cuando ha terminado el proceso de análisis, en el predicado analizar se realiza el

tratamiento de escritura de la información obtenida. Para ello se ejecuta el predicado *muestraAcordes*, definido en el archivo salida.pl, que recibe la lista resultado del análisis y se encarga de mostrarlo en el archivo de depuración. Junto a la definición de de ese predicado, se encuentran otros que asocian los enteros utilizados en la codificación interna del módulo de análisis con las cadenas de texto que luego son utilizadas en la clase de constantes, asociando enteros con su representación en lenguaje natural. Esto facilita enormemente la depuración del módulo, ya que en el fichero debugProlog.txt se encuentra información detallada sobre la secuencia de análisis reconocida, al final del proceso, pero también de otros pasos que durante el mismo se han realizado.

Tras cambiar la salida hacia el archivo salidaProlog.txt, se produce una llamada a ejecución del predicado *formatSalida*, que toma la lista de tuplas de nuevo y las escribe en el fichero de salida de tal manera que los enteros queden separados por espacios dentro de cada tupla. A su vez, cada tupla es escrita en una línea de texto, para que luego el parser de este archivo pueda distinguir de qué tipo de tupla se trata según el primer entero de la misma, como se ha explicado anteriormente. En esta etapa se presentaron problemas con la escritura retardada que Prolog hacía del archivo: fue necesario, además de usar el predicado *flush_output*, crear un archivo de testigo que, una vez completada la escritura de salidaProlog.txt indicara a la aplicación en Java que este archivo ya había sido completado, para proceder así a su tratamiento.

2.2.2. Lectura y procesamiento de ficheros midi

El formato MIDI

MIDI (Musical Instrument Digital Interface) es un protocolo creado en los años 70, estandarizado, que realiza la transmisión de los datos a partir de eventos y mensajes (y no de señales de audio). El formato MIDI describe una norma de comunicación física entre los sistemas (conectores, cables, protocolos de comunicación...) y las características del lenguaje que hacen posible el intercambio de información entre ellos.

Un archivo MIDI se puede entender como una partitura, ya que en sus mensajes está almacenados tanto cuándo tiene que sonar una nota como las características de dicha nota.

Dentro del formato MIDI existen varios tipos:

- SMF (Standard Midi File) 0: todas las pistas individuales se juntan en una única pista. Al cargar el archivo, por ejemplo, con un programa de notación, no se distingue a qué pista pertenece cada voz.

- SMF 1: En este caso las pistas permanecen separadas, lo que facilita su representación en una partitura.

- SMF 2: Es una ampliación del SMF 1, que cuenta con diferentes patrones de pistas separadas.

Bases para el tratamiento del archivo *.mid

El tratamiento del archivo MIDI ha tenido dos fases:

1. A partir de jMusic.
2. A partir de midi2abc (lenguaje ABC).

En primer lugar se planteó el uso de jMusic para leer los midis y obtener una representación de ellos que nos pudiera facilitar el análisis de la partitura. Esta decisión la tomamos cuando conocimos jMusic, ya que era una librería escrita en Java que nos facilitaba enormemente el tratamiento del midi: jMusic era capaz de leer un MIDI y devolver su partitura, formada ,entre otras características,por:

1. Una lista de las partes (voces).
2. El numerador del compás.
3. El denominador del compás.
4. Las alteraciones de la armadura.
5. El modo de la tonalidad (mayor/menor).
6. El tempo.
7. El título.

clases jMusic para la edición de partituras

La mayoría de estas propiedades nos era de gran utilidad en nuestro análisis. Sin embargo, después de un tiempo trabajando con jMusic nos dimos cuenta de que no tenía tanta utilidad como aparentaba, ya que el proceso de lectura del midi no lo hacía correctamente. Lo único que jMusic era capaz de obtener directamente del MIDI eran las pistas (partes en la partitura) y además no era del todo correcto, ya que el comienzo de las pistas no se fijaba bien (lo que repercutía directamente en nuestro análisis).

En un primer momento continuamos el trabajo procesando midis en los que comenzaran todas sus pistas a la vez, y fijando nosotros en código el comienzo de cada pista (todas a 0) para que el análisis no se descuadrara, pero esta solución no

nos permitía manejar midis en los que las pistas comenzaran en tiempos distintos (que eran la mayoría).

Como solución a este problema decidimos utilizar el programa midi2abc. Se trata de un programa que dado un fichero midi produce un archivo *.abc (se puede abrir con un editor de texto). La ventaja de ABC sobre jMusic es que las duraciones de las notas están mucho mejor definidas, además de que reconoce los instrumentos, compás, comienzo de pistas, título... Todas ellas son carácterísticas de ABC que al final hemos tenido que utilizar para hacer el procesamiento del archivo MIDI.

```
% input file G:/midis/bach\J.S.Bach-Catechism-Variation 1.mid
% format 1 file 5 tracks
X: 1
T:
M: 4/4
L: 1/8
Q:1/4=90
% Last note suggests minor mode tune
K:Bb % 2 flats
% Time signature=4/4  MIDI-clocks/click=24  32nd-notes/24-MIDI-clocks=8
V:1
%Brass Section
%%MIDI program 61
z8|z8|z8|z8|
z8|z8|z4 G3/2z/2 DG|_G=G/2A/2 DA B3/2z/2 G3/2z/2|
z4 G/2_G/2=G3/2z/2A|z/2z/2z/2z/2 AG/2A/2 B3/2z/2 Ge|=Bc/2d/2 G=B z/2c2z/2d|z/2z/2z/2z/2 _BA/2B/2 A3/2z/2 Dg|
_g=g/2a/2 dg cd/2e/2 _Gd|z/2z/2z/2z/2 z/2z/2A/2B/2 A3-A/2z/2|A3-A/2z4z/2|z4 Bc Bc|
z/2z/2z/2z/2 z/2z/2B/2c/2 de/2f/2 ed|cB _A=G ef/2g/2 fe|dc B=A f/2e/2f3-|fg/2f/2 ef/2d/2 z/2e3-e/2-|
ef/2g/2 f/2g/2e d4-|dg/2f/2 e/2d/2c/2B/2 A/2G/2F/2G/2 A/2B/2A/2B/2|c/2B/2A/2B/2 c/2d/2c/2d/2 e/2z/2d/2e/2 f/2z/2z/2g/2|z/2z/2z/2z/2 z/2c/2B B3-B/2z/
z8|z8|z8|z8|
z8|z4 d4-|dc ed d4-|df ge cf e/2d/2c|
Bc AB cF AB|cF fg AB AB|z/2z/2z/2z/2 BA/2B/2 c3/2z/2 F3/2z/2|z4 c3/2z/2 Fc|
de/2f/2 ed e3-e/2z/2|z4 c3/2z/2 Gc|fg/2_a/2 gf e4-|e3/2z/2 fd/2e/2 e3/2z/2 d2-|
db g_d c4-|ce =de/2c/2 g/2f/2g3/2z/2_a|z/2z/2z/2z/2 dc c3-c/2z/2|z8|
z8|z8|z8|z8|
z4 G3/2z/2 z/2_G/2=G|AB/2A/2 GA B3/2z/2 G3/2z/2|z4 G_G D=G|A_G DA B=G _G=G|
Dd B/2c/2A/2B/2 c/2d/2c/2d/2 z/2z/2d/2z/2|z/2z/2d/2e/2 fg/2e/2 e3/2z/2 d3/2z/2|z4 dB Gg|fg/2_a/2 dg e3/2z/2 dc|
z4 cA Ff|ef/2g/2 cf d3/2z/2 cB|z4 dc/2B/2 cd|GA/2B/2 ce/2d/2 ed/2c/2 de|
Ac/2B/2 cd/2e/2 _G3/2z/2 =ED|d4- de/2d/2 cd/2B/2|c4- cd/2c/2 Bc/2A/2|B4- Be/2d/2 c/2B/2A/2B/2|
c/2B/2A/2=G/2 A/2_G/2=G/2A/2 D/2=E/2D/2=E/2 _G/2=G/2_G/2=G/2|A/2G/2_G/2=G/2 A/2B/2A/2B/2 c/2z/2B/2c/2 d/2z/2z/2e/2|z/2z/2z/2z/2 AG G3-G/2
V:2
%Clarinet
%%MIDI program 71
z4 G,3/2z/2 D,G,|A,3/2z/2 D,A, B,D B,G,|_G,A, _G,D, =G,B, G,D,|E,D, E,C, F,A, F,C,|
D,C, D,B,, E,G, E,B,,|C,B,, C,A,, B,,3/2z/2 G,,B,,|D,3/2z/2 D,,3/2z/2 G,,3/2z/2 B,,3/2z/2|D,3/2z/2 _G,3/2z/2 =G,3/2z/2 D,G,|
A,3/2z/2 D,A, B,D B,G,|_G,A, _G,D, =G,3/2z/2 G,,3/2z/2|z2 F,3/2z/2 E,G, E,B,,|C,B,, C,A,, D,3/2z/2 D3/2z/2|
C3/2z/2 B,3/2z/2 A,3/2z/2 D,_G,|=G,3/2z/2 G,,G, _G,A, _G,D,|CB, CA, B,3/2z/2 =G,B,|C3/2z/2 A,C DF DB,|
A,C A,F, B,3/2z/2 B,,3/2z/2|z2 B,3/2z/2 C3/2z/2 C,3/2z/2|z2 C3/2z/2 DF DA,|B,A, B,G, C_E CG,|
A,G, A,F, B,D B,F,|G,F, G,E, F,3/2z/2 F,,3/2z/2|z2 F,3/2z/2 G,3/2z/2 D,E,|F,3/2z/2 F,,3/2z/2 B,3/2z/2 F,B,|
C3/2z/2 F,C DF DB,|A,C A,F, B,D B,F,|G,F, G,E, A,C A,E,|F,E, F,D, G,B, G,D,|
E,D, E,C, D,3/2z/2 B,,D,|F,3/2z/2 F,,F, B,D B,G,|_G,A, _G,D, =G,B, G,D,|E,D, E,C, F,A, F,C,|
D,C, D,B,, E,3/2z/2 E,,3/2z/2|z2 D,3/2z/2 C,F, E,C,|D,C, D,B,, F,3/2z/2 C,F,|G,3/2z/2 C,G, A,C A,F,|
=B,D =B,G, C3/2z/2 G,C|D3/2z/2 G,D EG EC|=B,D =B,G, CE CG,|_A,G, _A,F, _B,D B,F,|
G,F, G,E, _A,C _A,E,|F,E, F,D, E,3/2z/2 C,E,|G,3/2z/2 G,,3/2z/2 C,3/2z/2 G,,C,|D,3/2z/2 G,,D, E,G, E,C,|
=B,,D, =B,,G,, C,E, C,G,,|_A,,G,, _A,,F,, _B,,D, B,,F,,|G,,F,, G,,E,, _A,,C, _A,,E,,|F,,E,, F,,D,, E,,C,, E,,F,,|
G,,3/2z/2 G,,3/2z/2 C,,E, C,=A,,|_G,,A,, _G,,D,, =G,,3/2z/2 D,,G,,|A,,3/2z/2 D,,A,, B,,D B,G,|_G,=A, _G,D, =G,3/2z/2 D,3/2z/2|
```

ejemplo de programa ABC

Primeros pasos

Una de las dudas que tuvimos en un primer momento fue cómo comprobar que los ficheros midi que tratábamos los estábamos procesando correctamente. Como hemos indicado antes, existen distintos tipos de archivos MIDI, y el problema que teníamos era que no sabíamos cómo eran internamente nuestros midis de prueba (todos descargados de internet).

Este problema tuvo una solución sencilla (aunque nos ha llevado cierto tiempo) : crear nuestros propios ficheros midi. Para ello nos hemos ayudado de programas de edición de partituras, como son Finale o Guitar Pro, que además de permitir escribir una partitura en formato clásico dan la posibilidad de generar un MIDI a partir de ella. Gracias a estos archivos la localización de errores en el tratamiento ha sido relativamente simple (no así la solución de los mismos) y hemos podido comprobar que tanto la lectura de archivos MIDI como la creación de los acordes era correcta.

Algunos de los casos de prueba que realizamos son:

1. Pruebas de pistas, para comprobar que las distintas pistas se separaban correctamente en el análisis.

2. Pruebas con silencios al principio de la partitura, en una voz o en varias, anacrusas...

3. Pruebas para comprobar si los compases se obtenían bien.

4. Pruebas para comprobar la obtención de la tonalidad en Java/Prolog

Además tenemos que indicar que estas pruebas con midis creados por nosotros no se han realizado solamente en la fase inicial del proyecto, sino que nos hemos valido de nuestros propios midis a lo largo de todo el curso. Como ejemplo está la creación de midis con instrumentos definidos en cada pista para la comprobación de las etiquedas de los instrumentos en la interfaz.

Cómo se procesan los archivos *.mid

A continuación explicamos con más detalle el proceso que siguen los midis, desde que son leidos hasta que se genera el archivo entradaProlog.txt.

Lectura del fichero MIDI y generación del fichero ABC :
Como hemos indicado, para obtener la información del fichero MIDI hemos utilizado la herramienta midi2abc: se trata de un programa que dado un fichero *.mid genera un archivo *.abc, es decir, traduce el archivo midi a formato ABC.

La ejecución del programa midi2abc desde nuestra aplicación Java se ha hecho a través de una llamada al sistema operativo (igual que la llamada al analizador de Prolog).

Teniendo en cuenta que la sintaxis desde línea de comandos para ejecutar midi2abc es

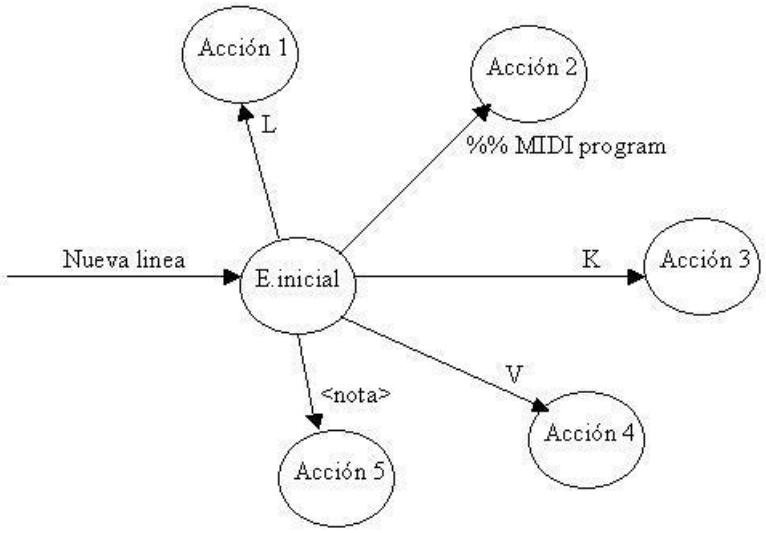
midi2abc -f ficheroEntrada [-o ficheroSalida]

la llamada realizada desde Java ha sido *Process p = Runtime.getRuntime().exec(cmd /C start midi2abc -f rutaMIDI rutaABC); p.waitFor();* Una vez realizado esto, en el directorio raiz del proyecto se encuentra un fichero *.abc que contiene el MIDI indicado traducido a formato ABC.

Tratamiento del fichero ABC para la obtención del objeto Score :
Una vez generado el fichero ABC, se procede al parseo del mismo. Para ello leemos el fichero y analizamos las líneas que nos interesan:

- Longitud de la nota más corta
- Armadura
- Nueva voz
- Instrumento
- Lista de notas

El parseo se puede representar mediante el siguiente autómata:



primeras decisiones del parser de ABC

- Acción 1: fijar la duración de la nota base.
- Acción 2: asignar el instrumento reconocido a la pista actual.
- Acción 3: fijar la armadura ABC. La omisión de esta acción descuadraría el análisis, ya que ABC no incluye las alteraciones en las notas que genera si la alteración correspondiente ya está indicada en la armadura.
- Acción 4: comienzo del reconocimiento de una nueva voz. En este caso:
  - Se añade la pista reconocida hasta ahora a la partitura.
  - Se crea una pista vacía.
  - Se activa el flag para reconocer el instrumento.
- nota: si se reconoce un carácer de comienzo de nota se lee la línea y se procesan las notas que se definen en ella. Los caracteres de comienzo de nota son:

$$a,b,c,d,e,f,g,A,B,C,D,E,F,G,\hat{},\_,=,[,z$$

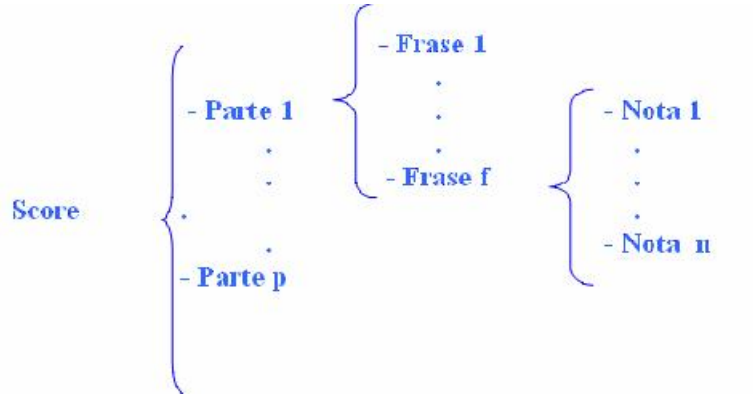
En el caso de reconocimiento de línea de notas el proceso es el siguiente:

- Se lee una o ninguna ocurrencia de los caracteres correspondientes al sostenido *(^)*, bemol *(_)*, becuadro *(=)* o acorde *([)*
- Se lee el nombre de la nota: *[a..g]* o *[A..G]*
- Se lee la octava en la que está la nota: caracteres , si la nota es *[A..G]* o ’ si la nota es *[a..g]*

Se comienza leyendo la cadena que da nombre a la notas

Tratamiento del objeto Score para obtener todas las notas :

Una vez que tenemos la partitura creada tenemos que procesarla para obtener las notas. Para ello podemos aprovechar la propia estructura del objeto Score de jMusic, que guarda los tracks en una lista de pistas. Como cada pista está formada por una lista de frases, necesitamos recorrer para cada pista su lista de frases y a partir de las frases, obtener las notas. La estructura que contiene las notas en sí se podría representar como:



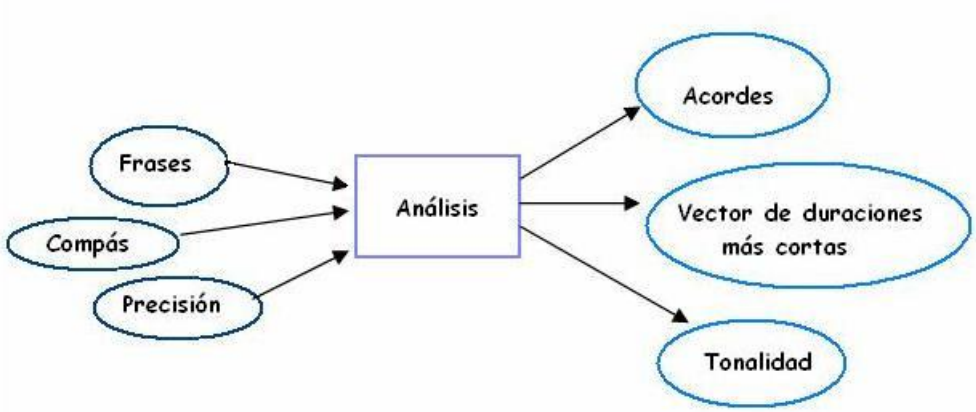
disposición de las notas en un objeto Score

Creación de los acordes y de los vectores de duraciones. Búsqueda de la posible tonalidad :

Esta es, probablemente, la parte del procesamiento MIDI más delicada. Para esta parte necesitamos varias variables contador que nos indiquen en qué voz estamos, en qué compás, pulso ... además, debemos tener en cuenta qué tipo de compás estamos usando. Una gestión incorrecta de estas variables podría provocar fallos en la creación de los acordes y que, por lo tanto, el proceso de análisis en Prolog fallara por no tener una base correcta.También es importante indicar el análisis se hace con un nivel de precisión fino (semicorchea, fusa...) para controlar el mayor cambio posible de acordes, y que posteriormente este análisis se simplificará para los casos en que sea posible unir acordes contiguos.

El proceso es el siguiente:

- Si estamos procesando la primera nota para el acorde, se crea un acorde nuevo.
- Añadimos la nota al acorde, indicando si es la parte fuerte del compás y/o la parte fuerte del tiempo (ambas cosas las son conocidas porque sabemos el compás de la partitura y llevamos contadores que nos indican las notas por compás y por tiempo que se han añadido).
- Al mismo tiempo, generamos los vectores que nos indican cuáles son las duraciones de las notas notas más cortas por compás y por tiempo. Estos vectores se usarán para la representación de la partitura.
- También a la vez buscamos cuál puede ser la tonalidad de la obra, a través del Algoritmo de Valoración de Alteraciones.



proceso de análisis inicial

Veamos ahora en qué consiste el algoritmo de valoración de alteraciones.

- En este algoritmo se procesa todas las notas de la partitura sin diferenciar voces. Para ello no es necesario hacer de nuevo un recorrido por las voces. En la creación del análisis incial, cada vez que manejamos una nueva nota para crear los acordes, realizamos además el tratamiento de la nota respecto al A.V.A.
- La base del algoritmo es un vector que lleva la cuenta del número de ocurrencias de cada nota.
- Cada posición del vector representa una nota.
- Cada vez que se procesa una nota, se incrementa en uno el contador asociado a esa nota. De esta manera, al terminar el recorrido de todas las notas de la partitura, tendremos guardada en cada posición del vector el número de ocurrencias que ha tenido su nota asociada.

0	1	2	3	4	5	6	7	8	9	10	11
Do	Do#/Reb	Re	Re#/Mib	Mi	Fa	Fa#/Solb	Sol	Sol#/Lab	La	La#/Sib	Si

vector del Algoritmo de Valoración de Alteraciones

- Al terminar de procesar las notas se comprueban los valores del vector.
  - Orden sostenidos: Fa, Do, Sol, Re, La, Mi, Si
  - Orden bemoles: Si, Mi, La, Re, Sol, Do, Fa
- Si hay más Fa sostenido que Si bemol, probablemente estamos en una tonalidad con sostenidos.

- Comprobamos si hay más Fa que Fa sostenido, Do que Do sostenido, Sol que Sol sostenido...
- Si hay más Sib que Fa sostenido, probablemente estamos en una tonalidad con bemoles.
- Comprobamos si hay más Si que Si bemol, Mi que Mi bemol, La que La bemol...
- Cuando llegamos a una posible tonalidad mayor, comprobamos si su relativo menor tiene más probabilidades de ser la tonalidad correcta. Para ello comprobamos si el número de veces que aparece el séptimo grado de la tonalidad menor es sensible o subtónica (medio tono o un tono de diferencia entre ella y la tónica).

Simplificación de los acordes : Tras el proceso de análisis inicial hemos conseguido una lista de acordes, todos de igual duración. El siguiente paso es unir los distintos acordes para obtener así, además de los acordes en sí, la duración real de cada uno. Con este fin, para cada acorde de la lista inicial de acordes:

1. Se compara cada acorde con el siguiente.
2. En caso de que sean iguales, se 'funden' los dos acordes, sumándole la duración del segundo a la duración que tenía el primero.

Generación del fichero de entrada a Prolog : Como último paso está la creación del archivo que servirá como entrada al analizador en Prolog. Este fichero debe tener una estructura concreta, para que el analizador en Prolog lo pueda interpretar correctamente. La estructura la podemos representar a través de una expresión regular:

fichero = [(tono,modo),(acorde)(,(cambioCompas,)? (acorde)+)]*

Es decir, una lista en la que el primer componente será siempre el par tono-modo que indicará la tonalidad de la partitura, seguido de una lista de uno o más acordes. Los acordes están separados por comas y en ocasiones, entre acorde y acorde, se genera una marca de cambio de compás.

Las distintas partes de la expresión regular se definen así:

tono = [0..11]. 0 representa la nota DO, 1 la nota DO#/REb, 2 la nota RE...
modo = [0,1]. 0 indica que el modo es mayor, 1 que es menor.
cambioCompás = 20.
acorde = duracion , [listaNotas]. Un acorde viene indicado, en primer lugar, por su duración, y en segudo lugar por la lista de notas que lo forman. En caso de que el acorde sea parte fuerte, su duración se ve incrementada en 20.
listaNotas = nota; listaNotas = nota , listaNotas es una nota seguida de una lista de notas, o simplemente es una nota. La nota viene representada por un enter [0..1], igual que en el tono.

2.2.3. Interfaz gráfica

La interfaz gráfica de la aplicación ha sido implementada en Java 1.5.05, concretamente usando Swing. A través de ella, el usuario tiene todo el control de la aplicación.
El usuario podrá abrir un fichero midi para ver su análisis, editar la partitura, actualizar el análisis, reproducir la partitura y consultar la ayuda. La interfaz es el módulo que tiene el control de la aplicación, y hace uso del procesador midi y del analizador cuando es necesario.

La interfaz está compuesta del paquete gui, aunque en ocasiones, usa las clases de los paquetes constantes y utilidades, que explicaremos más adelante. También hay que destacar que para cambiar el aspecto a la interfaz hemos usado distintos Look And Feels.

A continuación vamos a describir las clases que hay dentro de cada paquete:

Paquete gui

Es el paquete que contiene todas las clases relativas a la interfaz.

- BotonAcorde
  Se usa cuando estamos leyendo el resultado del análisis que nos devuelve el analizador mediante el fichero salidaProlog.txt. Se usa para almacenar toda la información necesaria para poder después dibujar todos los botones de los acordes de forma correcta. Como se puede ver por los atributos de esta clase, la información necesaria es el ancho que tiene que tener el botón, la fundamental, el tipo de acorde, el cifrado, la duración, el grado, el grado ajeno y una cadena de caracteres adicional para cuando sea un acorde ajeno, el tooltip que queremos poner al botón cuando lo dibujemos, el número de botón y el número de compás en el que está el botón.

- BotonAnalisis
  Contiene la misma información que la anterior, salvo que además tiene un atributo booleano llamado *botonAcorde* que si es true nos indica que el botón muestra el grado del acorde. Si es false, nos indica que el botón muestra la inversión del acorde. Este booleano tiene su importancia, porque los botones de grado tienen borde y los de inversión no, y a la hora de mostrarlos en pantalla hay que tenerlo en cuenta.

- FiltroFicheros
  Hereda de la clase FileFilter que viene incluida en Java. Esta clase sirve para que, cuando se abre un cuadro de diálogo para abrir o guardar un fichero, sólo se muestre el tipo de ficheros que nos interesen. En nuestro caso, los únicos ficheros que nos interesan son aquellos que tienen como extensión .mid.

- MuestraMensaje
  Es una clase muy simple, pero muy usada en la implementación de la interfaz. Lo único que hace es mostrar un mensaje del tipo que queramos. Su constructor tiene los siguientes parámetros: el mensaje que vamos a mostrar, el título de la ventana del mensaje y el tipo del mensaje. El tipo puede ser error, información y confirmación. El mensaje que muestra es un JOptionPane.

- PanelPartituras
  Es el panel que nos muestra las partituras del fichero midi que abramos. Hay que indicarle qué compases queremos que nos muestre porque mostramos la partitura paginada, para disminuir el consumo de memoria, ya que sería muy costoso dibujar a la vez y por tanto tener en memoria, la partitura entera de un fichero muy largo. Es una modificación de la clase jm.gui.cpn.Notate incluida en el JMusic. PanelPartituras se encarga de crear los pentagramas a través del objeto Score que le pasemos y mostralos por pantalla.

- VentanaAyuda
  Implementa la ventana con la que podemos consultar la ayuda de la aplicación. Tiene un JEditorPane que, básicamente es un editor capaz de interpretar y mostrar código html. Para que funcionen los enlaces de las páginas html de la ayuda, ha sido necesario incluir un HyperlinkListener y cuando se produce este evento, indicarle explícitamente al JEditorPane que cargue el documento html.

- CambioLF
  Sirve para cambiar el LookAndFeel (L&F) de la aplicación. El Look And Feel es el aspecto o apariencia de la aplicación, la forma que tienen los botones, los colores de los menús y las ventanas, el estilo de los cuadros de diálogo, etc. Cambiando el L&F podemos cambiar totalmente el aspecto de la aplicación. Esta clase tan solo tiene un método estático que recibe como parámetros la ventana principal y el número que representa el L&F seleccionado a través de la lista desplegable de la interfaz. Hemos utilizado cinco L&F diferentes. En ningún momento la aplicación toma el aspecto que viene por defecto con Java. Los L&F que hemos usado son gratuitos para uso no comercial y son los siguientes:

  - Nimrod
    http://personales.ya.com/nimrod/index.html
  - Tiny
    http://www.muntjak.de/hans/java/tinylaf/
  - Squareness
    http://squareness.sourceforge.net/skins/jlf.html
  - Substance
    https://substance.dev.java.net/
  - InfoNode
    http://www.infonode.net/index.html?ilf

  El cambio de L&F en ejecución se hace poniendo la siguiente instrucción, donde *ventana* es la ventana principal de la aplicación:

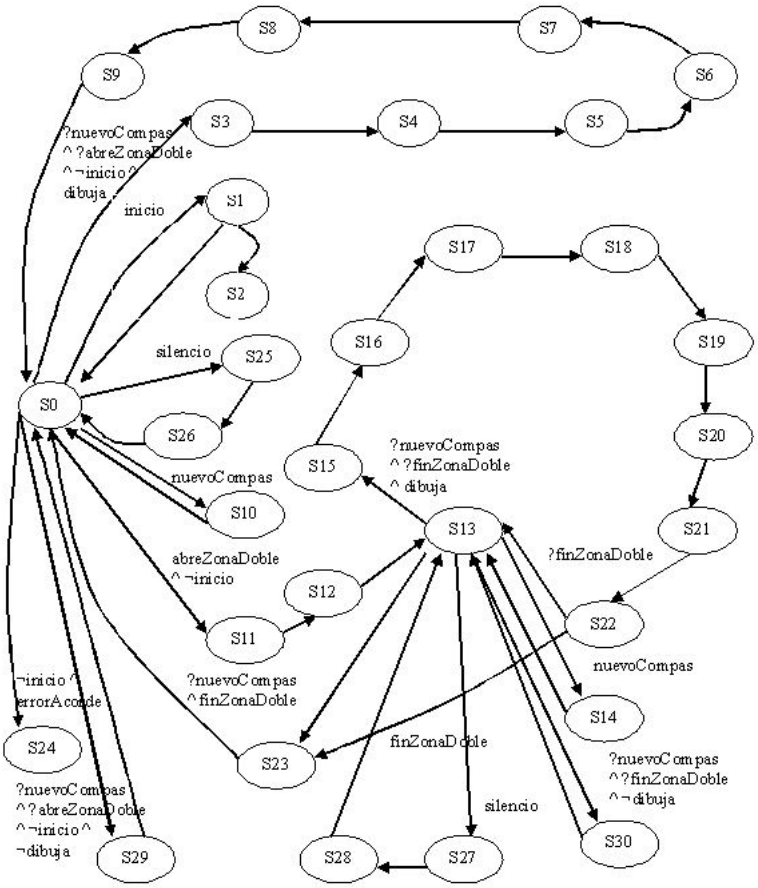
  ```
  SwingUtilities.updateComponentTreeUI(ventana);
  ```

- IVentanaPrincipal
  Forma parte de la ventana principal, junto con sus clases hijas AbstractVentanaPrincipal y VentanaPrincipal. IVentanaPrincipal inicializa la ventana, es decir, crea los menús, los paneles, los botones de acción, etc., inicializándolo todo.

Los métodos de acción como *abrirMidi*, *analisis*, *salir*, etc., son abstractos y están implementados en su clase hija.

- AbstractVentanaPrincipal
  Es la clase hija de IVentanaPrincipal y en ella se implementan los métodos de acción, es decir, aquellos métodos que son ejecutados cuando el usuario presiona un botón de acción de la interfaz como el botón para abrir midi, actualizar el análisis, consultar la ayuda o anterior y siguiente al visualizar una partitura. Es una clase abstracta, al igual que la anterior, cuya clase hija es VentanaPrincipal. Uno de sus métodos abstractos es *rellenaPanelAcordes* que lo comentamos más abajo.

- VentanaPrincipal
  Al igual que la clase anterior se encarga de ejecutar las acciones que solicita el usuario, VentanaPrincipal se encarga de realizar los cambios visuales. Por ejemplo, cuando se abre un nuevo fichero midi o cuando se actualiza un análisis, el análisis que se muestra por pantalla tiene que cambiar. Es decir, se encarga del contenido dinámico de la interfaz. Pues bien, la clase anterior, cuando el usuario quiere hacer una de estas dos cosas, llama al método *rellenaPanelAcordes* después de seleccionar un fichero y de llamar al procesador midi, que genera entradaProlog.txt y al analizador, que genera salidaProlog.txt usando entradaProlog.txt como entrada. Pues bien, este método, *rellenaPanelAcordes*, lee salidaProlog.txt a través de un autómata finito, siguiendo los pasos necesarios para interpretar correctamente la información del análisis, tal y como se explica en la parte del analizador. El autómata es el siguiente:



Autómata que lee salidaProlog.txt para mostrar el análisis

En el estado 0 del autómata no se realiza ningún cómputo. Veamos ahora el trabajo que realiza cada estado:

1. Se lee la fundamental de la obra.
2. Se lee la tonalidad de la obra.
3. Leemos la fundamental del acorde actual.
4. Leemos el tipo de acorde que es el acorde actual.
5. Leemos el cifrado del acorde actual.
6. Leemos la duración del acorde actual.
7. Leemos el grado del acorde actual.
8. Leemos el modificador del acorde actual.
9. Llama al método *dibujaZonaSimple* para que se muestren por pantalla los botones de los que se ha leído la información. Se incrementa el número del botón.

10. Si la variable *dibuja* es `true`, dibujamos los botones de los acordes y metemos separadores en todas las cajas que contienen los botones de acordes, tonalidad, duración, etc., porque en este estado, entramos cuando nos encontramos un nuevo compás en salidaProlog.txt. También incrementamos *numeroCompas* y actualizamos *dibuja* y *dibujaAnt*.
11. En este estado entramos, como se puede ver en el autómata cuando hay una zona doble, es decir, una zona con modulación. Si *dibuja* es `true`, ponemos *hayZonaDoble* a `true` y llamamos al método *dibujaBotonesAcordesZS* para que se dibujen todos los botones que queden por dibujar de la zona anterior sin modulación. Además leemos la fundamental que vamos a mostrar para la modulación.
12. Leemos el tipo de acorde que vamos a mostrar para la modulación.
13. Si *dibuja* es falsa y *dibujaAnt* es cierta, se dibuja la flecha (o línea) que abarca a todos los botones afectados por la modulación.
14. Estado que se ejecuta cuando nos encontramos un nuevo compás en salidaProlog.txt en una zona con modulación o zona doble. Es muy similar al estado 10.
15. Leemos la fundamental en una zona con modulación.
16. Leemos el tipo de acorde del acorde actual en una zona con modulación.
17. Leemos el cifrado en una zona con modulación.
18. Leemos la duración en una zona con modulación.
19. Leemos el grado en una zona con modulación.
20. Leemos el grado ajeno.
21. Leemos el modificador en una zona con modulación.
22. Dibujamos los botones de la zona doble.
23. Dibujamos la línea que afecta que abarca los botones de las acordes a los que les afecta la modulación.
24. Ha ocurrido un error en el análisis. Paramos el ánalisis y mostramos un botón de acorde cuyo texto es `error`.
25. Nos encontramos un silencio en una zona sin modulación. Asignamos a *fundamental* y *tipoAcorde* la cadena vacía y leemos la duración del silencio de salidaProlog.txt.
26. Dibujamos el botón de silencio.
27. Nos encontramos un silencio en una zona con modulación. Hacemos lo mismo que en el estado 25.
28. Dibujamos un botón de silencio de una zona con modulación.
29. Ignoramos la entrada porque no hay que dibujar. Se incrementa el contador en seis unidades para leer la información del siguiente acorde. La razón de que exista un estado como este es que, cuando *dibuja* es `false`, podemos ignorar mucha información ahorrando así mucho trabajo a la aplicación. El incremento del rendimiento es muy considerable gracias a este estado.

Paquete constantes

Es el paquete que contiene las constantes necesarias en la interfaz.
Sólo consta de una clase llamada constantesAnalizadorInterface que sirven para que el código del método *VentanaPrincipal.rellenaPanelAcordes* sea mucho más legible, porque en vez de usar enteros para referirnos a los posibles valores que pueda contener salidaProlog.txt, usamos nombres significativos para esos valores, que coinciden con los nombres de las constantes de esta clase.

Paquete utilidades

- AreaTexto
  Hereda de JTextArea y se diferencia de ésta en que, al llamar al método *imprime*, se añade una línea al cuadro de texto y además se escribe en la salida estándar si la variable booleana *ambos* es cierta.

- LectorFicheroAnalisis
  Sirve para leer, en este caso, el fichero salidaProlog.txt y tansformarlo en un array de caracteres que luego transformaremos en un vector de enteros para de esta manera, tener en cada componente de dicho vector, uno de los enteros que hay en salidaProlog.txt

- ExcepcionAnalizador
  Simplemente se usa cuando se va a abrir un fichero midi, y al aparecer el cuadro de diálogo para seleccionar el fichero, le cerramos o le damos a cancelar. Se usa para distinguir este caso de otros posibles errores y no tratarlo como un error.

Capítulo 3

Incidencias en el desarrollo

En esta parte, vamos a hablar sobre las dificultades con que nos hemos encontrando a lo largo de todo el desarrollo, cómo se han ido solucionando y la ayuda que hemos recibido de otras personas.

3.1. Problemas y soluciones

Uno de los principales intereses que había en la librería jMusic era su capacidad para mostrar partituras, algo que no se había encontrado en otra librería. Los resultados parecían ser bastante interesantes, pero el panorama cambió conforme nuestro desarrollo evolucionaba. El primer problema lo tuvimos al darnos cuenta de que los compases no aparecían alineados en vertical, de tal manera que la línea de fin de compás no coincidía en cada una de los pentagramas representados. Puestos en contacto con los desarrolladores de jMusic, se nos indicó que eso no tenía solución y que habría que realizar el proyecto desde cero para solucionarlo. Sin embargo, un análisis del código utilizado nos permitió llegar a una solución realmente útil. Los tipos de pentagramas que jMusic puede representar son: un pentagrama con clave de Fa (BassStave), un pentagrama con clave de Sol (TrebleStave), un sistema de dos pentagramas para la escritura pianística (PianoStave) y algo que pretendía ser una modificación para algún instrumento o agrupación instrumental con un ámbito extenso, pero que luego ni siquiera es contemplado en las clases de la librería (GrandStave). Estas cuatro clases derivan de una clase llamada Stave, donde se definen algunos métodos comunes a ellas.

El método *paint()* de cada clase estaba escrito para, a partir de la información almacenada en la frase asociada al pentagrama ir colocando las imágenes en la representación gráfica. Así, se dibuja primero las líneas del pentagrama, luego la clave o claves correspondientes, la armadura y la indicación de compás y finalmente empieza a colocarse nota por nota. El principal problema es que cada clase utiliza su propia definición del método *paint()*, lo que hace que haya mucho código repetido entre las clases, de tal manera que una modificación en ello sea muy costosa. Nuestra primera solución fue incorporar todo código común a las clases derivadas de Stave en ésta, utilizando una gran definición del método *paint()* de esta clase para proceder a la colocación de las notas. Como hay información que no puede ser compartida entre las clases, se definió una variable protegida de tipo entero en la clase padre de tal forma que en el constructor de cada clase hijo se almacenara para la instancia

correspondiente de qué tipo de pentagrama se trataba. Así, dentro de *paint()* se podía distinguir en aquellos casos en los que fuera necesario qué tipo de pentagrama era el que se estaba pintando. Esto era especialmente necesario en el caso de pintar las claves, por ejemplo. La colocación vertical de las notas según su altura y la clave o claves puestas a comienzo del pentagrama o sistema fue resuelto utilizando un método cuya cabecera estaba puesta en la clase Stave, obligando a cada clase hija a su definición. Así cada clase hija controlaba, por ejemplo, el pintado de líneas adicionales cuando la nota correspondiente excedía la tesitura del pentagrama.

El unir todo el código existente en una única clase nos ayudó a superar otro problema: puesto que los desarrolladores de jMusic estaban trabajando en cada clase por separado, había características que no estaban implementadas en todas ellas. En concreto, la clase TrebleStave era la más avanzada: controlaba la presencia de alteraciones accidentales y de las presentes en la armadura, de tal forma que se cumplían las normas de indicación de alteración del lenguaje musical. En las otras clases esto no se realizaba, y podía llegar a aparecer la misma nota dos veces sostenido en un mismo compás. Pero la principal mejora se obtuvo al aprovechar el código en TrebleStave que controlaba el mostrar ligaduras. Cuando en la información de la frase musical se encontraba una nota cuya duración excedía lo que restaba al compás, la clase TrebleStave era la única que tenía el código destinado a considerar qué valor de esa nota cabía en el compás actual y dejar el resto para el siguiente, pintando la correspondiente ligadura entre ellas. Cuando esto se producía en un pentagrama con clave de Fa, por ejemplo, la barra de ese compás no aparecía, de tal forma que había dos compases unidos sin estar separados por barras divisorias.

Mediante el reunir todo el código posible en la clase Stave se consiguió una gran clase, con un código especializado que era aplicado en casos particulares de sus clases hijas, quedando éstas de un tamaño mínimo, para albergar el constructor, con el que se establece en aquellas variables de la clase padre la información específica según el tipo de pentagrama y también el método para calcular la posición vertical en píxeles de una nota dentro del pentagrama, al depender esto de la clave utilizada. Esta modificación sobre el código de jMusic facilitó enormes mejoras: por un lado, el código era más fácil de interpretar y de modificar, como nos fue necesario más adelante para, por ejemplo, hacer la representación de los títulos al comienzo del pentagrama. Y por otro lado, el código era de mucha más calidad al evitar el reutilizar los mismos fragmentos en distintas clases que heredan de una misma.

El siguiente problema que solucionamos con la librería jMusic era el de que los compases no estaban alineados verticalmente. Esto es especialmente necesario en una aplicación que va a realizar un análisis armónico, ya que se pretende que el usuario sea capaz de comprobar que el análisis es correcto, de tal forma que pueda ver la coincidencia vertical de las notas que son reconocidas como integrantes de un acorde. Los desarrolladores de jMusic nos habían advertido de que no tenía solución, pero efectivamente no podía resolverse teniendo la concepción que ellos mantenían: había un método definido en la clase Stave que era llamado para cada nota, de tal forma que devolvía el ancho en píxeles que esa nota debía ocupar hacia su derecha, añadiéndose esa longitud al contador interno de la representación gráfica para que el siguiente símbolo musical fuera colocado en una nueva posición. El problema que tenían los desarrolladores era considerar siempre la misma distancia: no se podría conseguir una coincidencia vertical mientras todo valor presente en cualquier compás

tuviera una longitud en píxeles igual.

La solución que desarrollamos para esto fue compleja: por un lado, a la hora de procesar el fichero midi de entrada se obtuvo en un vector de enteros el valor mínimo de las notas de cada compás. Por otro lado, se redefinió el método que asignaba una longitud horizontal a cada nota: las longitudes no mantenían la misma relación que las duraciones correspondientes (por ejemplo, la longitud horizontal asignada a una corchea no coincidía con el doble de la de una semicorchea), ante lo que tuvimos que resolver esto. Sin embargo, no era una solución suficiente: se ha utilizado un vector estático, *duracSemiQuaver* que define la longitud horizontal en píxeles asignada a las semicorcheas de cada compás. Este vector almacena en la posición i el ancho asignado a la semicorchea del compás cuya nota de menor valor tiene duración i. Aplicando entonces multiplicaciones por dos a las duraciones a partir de la semicorchea conseguíamos que las voces pudieran tener valores distintos pero que hubiera coincidencia vertical en cada una de las partes del compás y sus subdivisiones. Además, ante un cambio de compás, se accede a *duracSemiQuaver* para tomar la longitud de la semicorchea según el mínimo valor de las notas del compás, de tal forma que los compases tengan ancho variable según el valor más pequeño de las notas en su interior. De esta forma, ya se garantiza que hay una coincidencia vertical de las notas, lo que provoca un poder leer acordes en vertical y, lo que es más interesante, poder situar encima de cada acorde la información asociada a él que se ha obtenido en el análisis.

Un nuevo problema ocurrido con jMusic fue darnos cuenta de que sólo mostraba hasta cierto número de compás, a partir del cual la representación gráfica se cortaba. Analizando el código de la librería nos dimos cuenta de que las notas eran almacenadas como ficheros .gif que, a la hora de construir la partitura, eran colocadas sobre una imagen en memoria, en la posición correspondiente a la altura que se pretendía representar. Esta imagen adicional luego era pintada en pantalla utilizando técnicas de doble buffering. Y precisamente ahí estaba el problema: el desarrollador no había contemplado la posibilidad de partituras que superaran cierto límite, el impuesto por el tamaño de imagen definido para el doble buffer, lo que provocaba que cuando en la representación de una partitura se llegaba al final de la imagen reservada en memoria para la colocación de las notas, no se mostrara nada más. Hicimos pruebas aumentando el tamaño del doble buffer de tal forma que cupiera toda la partitura, pero esto nos llevaba a frecuentes problemas de *OutOfMemoryException* cuando la partitura a representar era de cierta longitud.

La solución que pensamos entonces fue usar paginación: colocar botones en nuestra aplicación que facilitaran al usuario pasar de una página de la partitura a otra. Y esto suponía a su vez una modificación del código de la representación gráfica de la partitura que, ahora sí, ya estaba unificado bajo la clase Stave. Para cumplir este cometido tuvimos que definir unas variables que permitieran llevar la cuenta de compases. jMusic no llevaba un contador así, y el entero que indicaba el compás se obtenía como el resto de dividir el pulso actual dentro de la frase entre las cuatro negras que por defecto aplicaba a todo compás. Para decidir qué compases eran colocados en cada página se ha definido una constante a partir de la cual, al abrir un fichero midi, se calculaba el ancho en píxeles que va a ocupar cada compás en la representación gráfica. Esto es especialmente fácil si tenemos resuelta la alineación vertical de los compases, como se ha explicado en un apartado anterior. Calculado

el ancho, se utiliza una constante definida para saber cuántos compases se muestran por página: se consideran tantos compases por página como la suma de sus anchos no exceda el ancho en píxeles asignado a cada página.

A nivel de código de Stave, una vez más, la modificación era fácil debido a que se había reunido el código en una sola clase. Hubo que usar una variable para ir contando el número de compases. En cada llamada al método *paint()* de cada pentagrama, el código de jMusic lo que hace es recorrer todas las notas de la frase musical y las va colocando en el doble buffer. Se tuvo que añadir la comprobación para que sólo fueran añadidas a la imagen virtual aquellas notas que se sitúen los compases correspondientes a la página actual de pintado, lo que se consigue fácilmente usando sentencias condicionales. De esta manera, cuando el usuario presiona el botón para avanzar o retroceder página, se cambia en la clase Stave unos enteros que indican desde qué y hasta qué compás representar y en la ejecución del método *paint()* esta comprobación se hace constantemente.

El uso de paginación solucionó los principales problemas: el consumo de memoria se redujo drásticamente al usar sólo un fragmento de la representación gráfica que queríamos, además de poder controlar el tamaño asignado a la imagen del doble buffer. También la interfaz gráfica quedó mucho más agradable al usuario, al no tener que usar tanto las barras de desplazamiento para visualizar una partitura y poder desplazarse por ésta usando páginas.Sin embargo, llegó a otro problema: la edición de la partitura. jMusic tiene una clase asignada al tratamiento de eventos que ocurren sobre los pentagramas, llamada StaveActionHandler.

Cuando con el ratón se pincha sobre una nota y se desplaza, esta nota es modificada (ya sea en altura o en duración). La manera que tiene el código de saber sobre qué nota se ha hecho clic es ir almacenando en un vector el número de nota en la frase junto a la posición horizontal en píxeles a la que ésta se sitúa. Después, tomando del evento del ratón la posición horizontal en la que ha ocurrido, se recorre ese vector buscando la nota correspondiente, tomando la posición en la que se encuentra y modificando entonces en la frase la nota en la misma posición. Al usar paginación, esto sólo funciona en la primera página, ya que en las siguientes las notas están desplazadas respecto a su posición total en la frase. Para solucionar esto hubo que implementar un contador de notas, de tal manera que para cada nota se supiera cuántas iban antes que ella, en la página y así, a la hora de detectar qué nota se había modificado, se sumara el número de notas que había antes de la actual a la posición obtenida en el vector correspondiente, y así modificar la nota adecuada en la frase musical.

Con la edición también se detectó otro problema: el modificar la duración de las notas provocaba que todo se desencajara verticalmente, ya que la voz correspondiente en la que se realizaba la modificación llegaba a ocupar más o menos. Para solucionar esto, se ha desactivado el tratamiento del evento que ocurre cuando el usuario arrastra horizontalmente el ratón, intentando modificar la duración de las notas. Sólo se permite modificar la altura de una nota existente, manteniendo su duración. Ocurre algo parecido con el añadido de las notas: en el tratamiento de un evento, si el clic se ha hecho en una zona en la que no hay notas, se añade una nueva en la frase musical en la posición relativa al clic dentro de la partitura, desplazando a las siguientes. Esto, de nuevo, lleva a un desajuste de las voces, máxime cuando no se permite el borrado de una nota añadida, lo que puede llevar a equivocación

al usuario. También se ha desactivado esta posibilidad en la edición, de tal manera que sólo se pueden modificar las notas ya existentes.

Otras dos características desactivadas en la edición fueron la posibilidad de modificar la armadura, ya que no tiene sentido facilitarla. Puesto que el módulo de análisis reconoce la tonalidad de la obra y la información de la armonía se facilita respecto a ésta, no parece tener sentido permitir al usuario modificar la armadura presente en cada pentagrama, que ha sido puesta por la parte en Java según la información recibida por el módulo en Prolog. También había un problema con la indicación de compás. jMusic por defecto sólo admitía compases de subdivisión en negras, aunque el número de éstas en cada compás podía variar de uno a nueve. Se tuvo que modificar el código para contemplar, como hace la especificación midi, compases en los que la indicación del denominador sea una potencia de dos y el numerador sea cualquier número posible. Esto fue rápido, una vez más, gracias a la unificación de código bajo la clase Stave, aunque hubo que cambiar parte del código por el que jMusic llevaba a cabo el cálculo de cuándo el compás se acaba, así como cuándo es necesario partir una nota en dos usando una ligadura.

Otro problema más de esta larga lista fue que las clases de JMusic que servían para representar la partitura de forma gráfica estaban implementadas usando clases de AWT, y no de Swing. Swing es un conjunto de clases mucho más fáciles de usar que AWT, y actualmente, su uso es mucho más extendido. Como nosotros para implementar la interfaz gráfica usamos Swing, había que pasar todo el código de JMusic que participaba en la representación de las partituras de AWT a Swing, debido a que ambas librerías, aunque comparten algunas clases, no son del todo compatibles y no coexisten fácilmente. Hacer esta transformación, parece una tarea bastante fácil, pero nos dio más problemas de los que esperábamos. El panel donde mostramos las partituras, que internamente, en la aplicación, es independiente del panel de análisis, es una modificación de la clase Notate de JMusic. La clase Notate de JMusic, al llamar al constructor, mostraba una ventana desde la que se podían abrir ficheros midi para mostrar su partitura, así como muchas otras funciones. Como la ventana principal de nuestra aplicación iba a ser muy distinta a la de JMusic, nos interesaba que esta clase, que se encargaba de representar las partituras del midi abierto y creaba cada uno de los pentagramas(Staves), fuera un panel y no una ventana como era inicialmente. Al principio hicimos que PanelPartituras, que así se llama nuestra clase, heredase de Panel en lugar de Frame. Luego incluiríamos PanelPartituras en nuestra ventana principal. Al intentar que PanelPartituras fuese un JPanel y no un Panel, había pentagramas que no se mostraban, se lanzabn excepciones, etc. Como es lógico hubo que cambiar muchas clases para que todo funcionase correctamente. Se pasaron de AWT a Swing todas aquellas clases que tenían algo que ver con la representación de la partitura, siendo estas clases numerosas y con funciones muy diferentes.

También hubo problemas al abrir y guardar midis. Cuando abrimos un fichero midi con JMusic, se crea un objeto Score que almacena toda la información que necesitamos del fichero. Score representa la partitura y contiene las partes, las frases, las notas, etc. El problema era que, al pedir a JMusic el tiempo de inicio de cada pista, algunas aparecían desplazadas, lo que provocaba que todo el análisis que se realizaba después fuera totalmente incoherente: las pistas se alineaban entre sí de manera errónea, por lo que el análisis vertical (análisis armónico) no era el correcto.

La solución que primero se nos ocurrió fue intentar localizar en el código del MIDI el comienzo de cada pista, y asignarle instante de comienzo correcto, pero esta solución no era viable, ya que ponernos a manejar el flujo de bytes del midi con este fin lo veíamos completamente imposible. Como segunda solución pensamos hacer lo siguiente: ya que el problema estaba solo al comienzo de cada pista, pero a partir de ahí parecía que la medida se mantenía bien, podíamos intentar insertar una nota al comienzo de todas las pistas, para asegurarnos de que todas comenzaban en el instante 0.0 de tiempo. Esta solución se intentó implementar a través del programa *abc2midi*. El proceso era el siguiente: leíamos un fichero MIDI ejecutando *midi2abc*, al fichero resultante le introducíamos una nota al comienzo de cada track, generábamos un nuevo midi gracias a *abc2midi* y por último procesábamos este midi. La idea era muy buena, pero jMusic resultó dar fallos en varios sitios, no solo en el startTime, y el análisis seguía descuadrándose. Como última opción se nos ocurrió tratar todo el fichero generado por ABC, parsearlo y convertirlo a un objeto Score para, a partir de ahí, poder continuar con nuestro análisis igual que lo hacíamos antes. Aprovechábamos así que los ficheros generados por *midi2abc* tenían una estructura rítmica mucho mejor definida que jMusic (como detalle, indicar que ABC genera barras de compás y jMusic no). De esta manera conseguimos finalmente una lectura de midis normal, sin sobresaltos a no ser que ABC incluyera en la generación algún carácter extraño (por ejemplo, dosillos, tresillos...).

Siguiendo con los problemas del jMusic, otro de ellos fue que no reconocía el compás de una partitura: siempre era 4/4. En un principio (antes de usar ABC) se realizaba una pasada por el código de bytes del midi, y se buscaba la cadena de bytes que indicaba el compás. En un principio funcionó correctamente, pero al comenzar a utilizar ABC comprobamos que éste también resolvía el compás. Evidentemente, entre realizar una pasada por el código de bytes del MIDI o leer una línea del fichero ABC en la que se indicaba el compás, nos quedamos con esta última solución (así también mejoraba un poco el rendimiento de la aplicación).

En cuanto a salvar los cambios que realizásemos en una partitura mediante un fichero midi, JMusic tampoco lo hacía correctamente. Este problema ha quedado sin solución debido a los numerosos contratiempos que hemos tenido con JMusic. Había problemas a los que les hemos dado mayor prioridad como son la alineación vertical de los compases o la lectura de ficheros midi. La solución habría sido pasar el objeto Score a notación ABC y ejecutar luego abc2midi para generar el midi correspondiente, y ya tendríamos la funcionalidad de salvar un fichero midi.

Y para terminar con los problemas de jMusic, señalar que éste no indicaba los instrumentos utilizados en cada pista (a pesar de tener un campo específico para ello). La solución volvió a ser el fichero ABC generado para procesar las notas. Gracias a ese fichero, y a un traductor que construimos que nos devolvía la cadena asociada al instrumento que representaba la nota, hemos podido mostrar, tanto en la partitura como el cuadro que solicita al usuario las pistas que quiere utilizar, los instrumentos seleccionados.

3.2. Ayudas y contactos

Una de las principales ayudas para el módulo de análisis fue la de Larry Solomon. Su teoría era conocida por el profesor director de este trabajo quien, ante el problema

que teníamos por todo el trabajo que hacía Prolog al reconocer acordes, nos propuso usar su teoría musical. Se ha explicado antes que el cálculo de la forma prime supone realizar una serie de cálculos numéricos con las notas de un acorde para obtener una secuencia numérica que identifica de forma unívoca cada tipo de acorde. Esto avanza mucho trabajo en el proceso de Análisis Nominal, de tal manera que facilitó las cosas. Sin embargo, el principal problema es que considera por igual las notas presentes en un acorde, independientemente del número de apariciones. Esto era problemático en el reconocimiento de acordes que tuvieran alguna nota añadida, ya que esta nota pasaba a tener igual importancia que las del acorde al quitar los duplicados.

Nos pusimos en contacto con él para explicarle la situación que habíamos detectado, y cómo en un acorde de Do Mayor, con las tres notas que lo forman varias veces presentes en un gran acorde, si se añadía un La, la forma prime devolvía que se trataba de un acorde de La menor con séptima diatónica. En concreto, el correo electrónico que le enviamos fue el siguiente:

Somos estudiantes de Informática que estamos desarrollando un proyecto sobre reconocimiento armónico. La idea es dar un fichero midi a la aplicación como entrada y entonces reconocer los acordes utilizados. Para restringir la búsqueda en Prolog, estamos tomando cada acorde y, a partir del cálculo de su forma prime, saber así de qué tipo de acorde se trata (por ejemplo, Sol Mayor).

El problema es que pensamos que el cálculo de la forma prime da igual relevancia a cada nota en el conjunto del acorde debido a la eliminación de duplicados. Supongamos que tenemos el siguiente conjunto de notas: Do Mi Sol Sol Mi Mi Sol Sol Do Do Sol Mi La. Esta nota La es una nota de paso entre dos, una del acorde anterior y otra del siguiente, por lo que La no pertenece al acorde de Do Mayor. Usando el cálculo de la forma prime, este acorde es reconocido como La menor con séptima en vez de Do Mayor con una nota adicional.

¿Qué nos aconseja que podemos hacer para evitar esto?

La respuesta de Larry Solomon fue muy rápida y quizás esclarecedora:

Habéis tocado uno de los principales problemas del análisis armónico que todavía no está resuelto, llamado la Teoría de la Partición, que consiste en cómo decidir qué notas de un acorde le pertenecen y cuáles no. No hay ninguna teoría generalmente aceptada sobre esto, pero creo que en mi website hay algo al respecto, en el apartado de MAS y Set Theory. Básicamente, pienso que es necesario realizar un análisis estadístico preliminar sobre cada acorde, para establecer la consistencia del lenguaje armónico, reduciendo los acordes con sus añadidos a unos pocos acordes esenciales que aparecen recurrentemente. Entonces, hay que buscar de nuevo esos acordes.

En la música tradicional, esta recurrencia está claramente instaurada: sabemos que hay unos tipos básicos de acordes que hay que buscar, establecidos por la tradición. Sin duda, la solución más simple (y más elegante) es la preferida, al igual que ocurre en la ciencia. Sin embargo, en muchas de las obras contemporáneas del siglo XX, por ejemplo en las de Schomberg, no hay establecido ningún lenguaje armónico. Establecer particiones en ese tipo de música es una pesadilla.

Por lo tanto, también está el problema de cómo explicar las notas adicionales de un acorde (aquellas que no pueden ser consideradas dentro de él). Una mínima parte de este problema está en la definición de tonos no armónicos (comprueba en mi web el ensayo sobre este asunto). Incluso en un análisis armónico tradicional, es posible

llegar a varios y diferentes, aunque todos correctos, resultados para la misma obra.

También nos pusimos en contacto con Luis Ángel de Benito Ribagorda, profesor de Armonía en el Real Conservatorio Superior de Música de Madrid. Fue quien nos sugirió que, ante el problema de cómo reconocer una tonalidad dadas las características de cómo se almacena la información musical en un fichero midi, contáramos la incidencia de las alteraciones: si estamos en una tonalidad determinada, deben aparecer mayoritariamente las notas que en esa tonalidad tienen una alteración, frente a la misma nota en estado natural. Recorriendo el círculo de quintas si, por ejemplo, aparece más veces la nota Fa sostenido que Si bemol, podemos pensar que estamos en una tonalidad con sostenidos y, a partir de ello, determinar de qué tonalidad se trata comprobando la presencia de las alteraciones de la misma forma. La sugerencia realizada por este profesor se mostró válida una vez que fue implementada en la parte en Java. Sin embargo, fue considerada como una de las tres heurísticas utilizadas para el reconocimiento de la tonalidad, ya que también queríamos considerar la tonalidad con mayor número de acordes reconocidos dentro de ella, para así minimizar el número de apariciones de una zona con modulación. Como se explicó en el apartado correspondiente, estos dos métodos se han demostrado igualmente válidos: durante las pruebas realizadas al proyecto se constató que casi en la totalidad de los casos había coincidencia en el resultado de las dos heurísticas.

Para solucionar los numerosos problemas que nos surgieron usando JMusic, decidimos ponernos en contacto también con los creadores de la librería. JMusic es un proyecto de investigación de la universidad *Queensland University of Technology*, de Australia, donde se usa para enseñar a alumnos graduados y no graduados composición entre otras cosas. Así pues, escribimos a Andrew Brown, Rene Wooler y Tim Opie, que son unos de los creadores de dicha librería y nos suscribimos a la lista de correo de JMusic para desarrolladores, cuya página de suscripción es: `https://lists.sourceforge.net/lists/listinfo/jmusic-dev`

Los contactos no fueron muy fructíferos, ya que por ejemplo, a Andrew Brown le comentamos las dificultades que teníamos para alinear los compases de las distintas voces verticalmente, y le preguntamos si había alguna forma de hacer que JMusic lo hiciera, ya que nos extrañaba que no fuera capaz de hacerlo. La respuesta de Andrew Brown fue literalmente que le parecía más fácil empezar a programar JMusic de cero que modificar el código de JMusic para que alinease los compases, respuesta que no invitaba al optimismo. Escribimos a Rene Wooler para ver qué nos decía él y nos contestó que él no se había encargado de esa parte del código y hablásemos con Tim Opie. Tim Opie nos dijo que le enviásemos el código que tuviéramos y le explicásemos detalladamente qué es lo que queríamos hacer. Le enviamos un prototipo de nuestra aplicación por aquella fecha y no conseguimos respuesta. También estuvimos informándonos en foros como los de `www.programacion.com/java` e `www.hispasonic.com` sobre la existencia de alguna librería para Java similar a JMusic para realizar con ella al menos las cosas que en JMusic no funcionaban, y tampoco obtuvimos respuesta. Nosotros tampoco encontramos ninguna librería gratuita similar. Además, como se ha explicado, había más problemas con JMusic para abrir y guardar midis, y con la lista de voces de un midi, que tampoco funcionaban. Al final optamos por modificar el código fuente de JMusic por nuestra cuenta y riesgo, sin tener garantías de que íbamos a tener éxito y usar ABC (midi2abc) para

abrir los ficheros midi correctamente y obtener la lista de instrumentos.

Otro problema que hubo en la interfaz, es que JMusic estaba implementado usando las librerías gráficas AWT en lugar de Swing. Las librerías Swing son más modernas y fáciles de usar. Había que cambiar en el código fuente de JMusic, todas las clases que participaban en la representación de las partituras de AWT a Swing, y nos llevó bastante tiempo solucionar un problema que, a priori, parecía bastante sencillo.

Capítulo 4

Manual de usuario

4.1. Requisitos

Nuestra aplicación es una aplicación sencilla que no requiere muchos requisitos para funcionar, ya que prácticamente todas las librerías y herramientas que necesita usar en segundo plano están incluidas en la aplicación.

Por ejemplo, para usar el analizador en Prolog, no hace falta tener instalado Prolog, sino que ya incluimos un ejecutable con algunas librerías `dll` necesarias para su correcta ejecución. Lo mismo ocurre con ABC o con JMusic. ABC está incluido en un ejecutable llamado midi2abc y JMusic está incluido como clases del proyecto.

El único software imprescindible para la ejecución de la aplicación es *Java Runtime Environment 5.0 update 5* o superior. Si además queremos modificar el procesador midi o la interfaz, necesitaremos instalar *JDK 5.0 update 5 o superior*:
http://java.sun.com/javase/downloads/index.jsp

Si quisiésemos modificar la parte en Prolog (analizador), deberíamos instalar el *SWI Prolog*:
http://www.swi-prolog.org

El equipo que necesitaremos para la ejecución de la aplicación debe tener al menos las siguientes características:

- Windows 98/Me/2000/XP o Linux
- Procesador Pentium II 300 MHz
- 64 MB de memoria RAM
- 20 MB de espacio libre en disco duro
- Recomendada resolución 1024x768
- Instalado *JDK o JRE 5.0 update 5*

4.2. Instalación

Para la instalación del Analizador Armónico, aparte de tener instalado *JRE* como hemos comentado antes, lo único que hay que tener es una copia de la aplicación. Una vez hayamos conseguido la copia, lo único que hay que hacer es guardarla en

nuestro disco duro en la carpeta que queramos y ya estará la aplicación lista para funcionar.

4.3. Ejecución y manual de uso

4.3.1. Ejecución

Ejecutar la aplicación es realmente sencillo, ya que disponemos de ficheros para lanzar la ejecución.

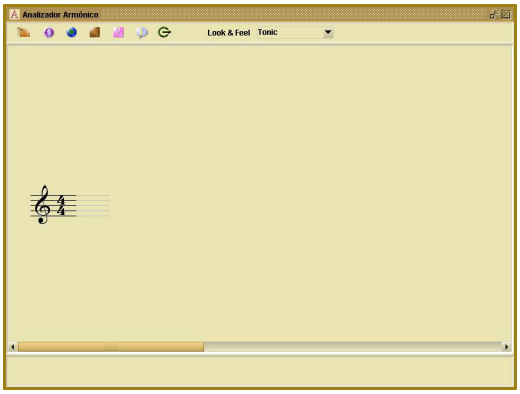
Si estamos en Windows, bastará con ejecutar ejec.bat desde la consola de comandos o haciendo doble clic sobre él.

Si estamos en Linux hacemos lo mismo, solo que con el fichero ejec.sh.

4.3.2. Manual de usuario

En el manual de usuario, veremos paso a paso cómo usar la aplicación. Hay que destacar que la aplicación tiene ayuda, que podemos consultar para cualquier duda que nos pueda surgir.

Nada más arrancar la ejecución con ejec.bat en Windows o ejec.sh en Linux, vemos la siguiente pantalla:



Pantalla principal de la aplicación

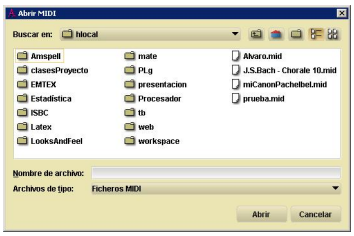
Como podemos ver, aparece una partitura vacía, y mucho espacio vacío en el panel principal. En ese panel aparecerá una partitura y el resultado del análisis cuando abramos un fichero midi.

La barra de tareas posee botones para interactuar con la aplicación:

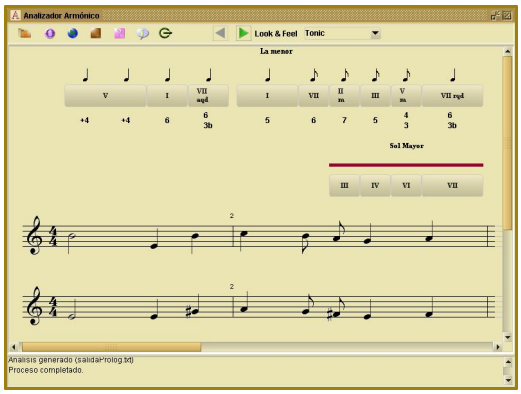
A continuación iremos viendo toda la funcionalidad de la aplicación:

Abrir Midi

El botón  es el que usamos para abrir un fichero midi. Cuando pulsamos sobre el siguiente botón, se dan una serie de pasos. Lo primero que vemos es un cuadro de dialógo como este para seleccionar el fichero que queremos abrir:



Una vez seleccionado el fichero que deseamos, la interfaz llama al procesador midi para generar entradaProlog.txt. Una vez generado este fichero, se muestra una lista de instrumentos (o de voces) que tiene el fichero midi, para que el usuario pueda elegir qué voces se consideran al realizar el análisis. Una vez seleccionados los instrumentos, se llama al analizador en Prolog para que genere salidaProlog.txt. Y por último, cuando el analizador ha terminado su trabajo, la interfaz lee salidaProlog.txt y muestra el análisis generado por pantalla, así como los pentagramas de las voces seleccionadas. Obtendríamos una pantalla similar a esta:



Podemos observar que, en la imagen anterior, aparece en el análisis una segunda fila de botones con una línea roja horizontal por encima. Esta doble fila de botones aparece cuando abrimos un fichero midi en el que se han detectado modulaciones.
Nos podemos mover por la partitura y el análisis usando los botones anterior y siguiente de la barra de tareas. 

Refrescar análisis

Cuando tengamos una partitura cargada, podemos modificar sus notas pulsando sobre ellas y arrastrando el ratón hacia la izquierda, derecha. Una vez hayamos cambiado alguna nota, podemos pulsar sobre el botón  para refrescar el análisis. Una vez refrescado el análisis veremos los cambios por pantalla.
Es posible que el orden de las voces cambie tras refrescar el análisis.

Reproducir una partitura

Podemos reproducir una partitura si hemos abierto antes algún fichero midi. Para ello usaremos los botones de `play` y `stop`:

Si tenemos una partitura cargada que hemos modificado, al presionar el botón `play` se reproducirá la partitura modificada, no la cargada inicialmente al abrir un fichero midi.

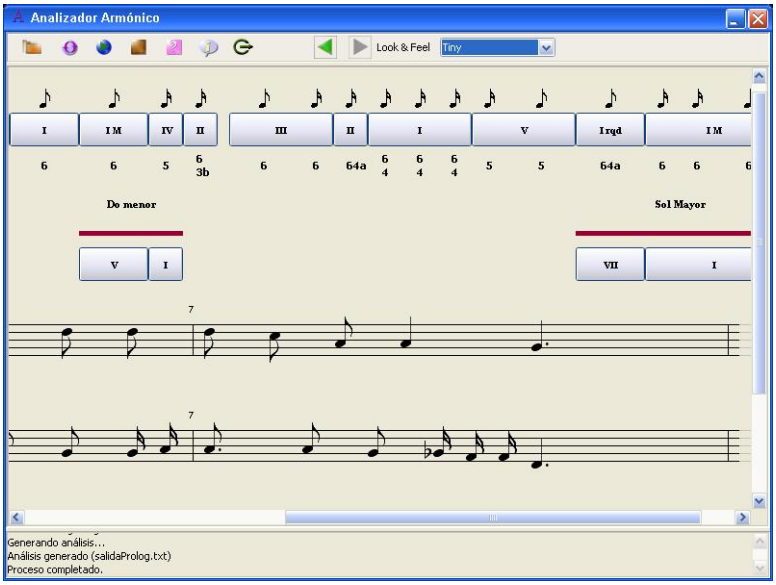
Consultar la ayuda

Hemos incluido ayuda en formato html para que el usuario pueda guiarse para usar la aplicación. La ayuda incluye una breve explicación de para qué sirve un analizador de armonía musical y qué es capaz de hacer, así como los pasos a seguir para usar correctamente la aplicación. Se puede navegar por la ayuda a través de los enlaces que tiene como si fueran páginas web.

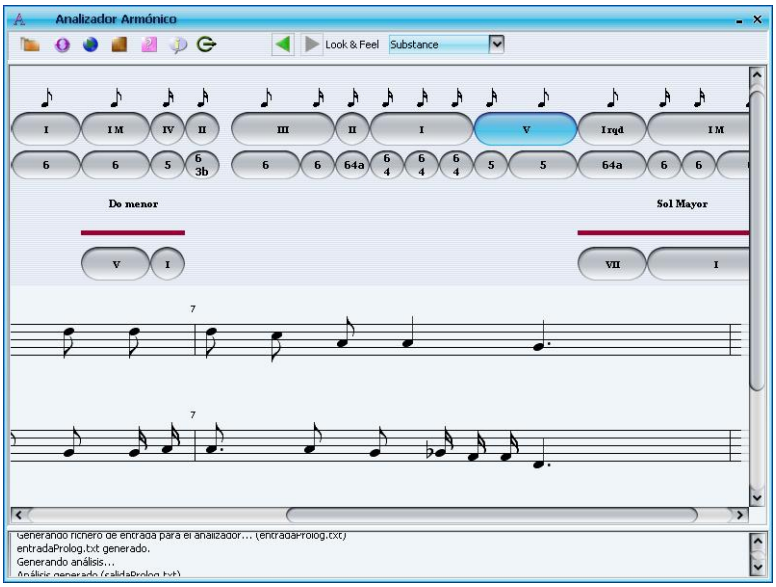
Ventana de ayuda de la aplicación

Cambiar el aspecto

Podemos cambiar el aspecto de la aplicación y trabajar con el que más nos guste o nos resulte más cómodo. Podemos hacerlo mediante la lista desplegable que está en la barra de tareas. Hemos incluido cinco aspectos diferentes. Por defecto el aspecto utilizado es `Nimrod`. Hay un aspecto imitando a Windows XP, que en Linux puede resultar curioso, e incluso otro imitando al famoso entorno Aqua de los ordenadores Macintosh. Estos dos aspectos quedan de la siguiente manera:



Aspecto con Tiny L&F



Aspecto con Squareness L&F

Capítulo 5

Características y mejora de la aplicación

A continuación vamos a describir las características de la aplicación, así como las mejoras que se podrían realizar, cómo podrían llevarse a cabo y cuánto esfuerzo costaría.

5.1. Características

El analizador armónico musical cumple la función para la que fue creado, que es realizar un análisis de la armonía de una forma "inteligente" y efectiva. Como comentaremos más adelante, hay muchas cosas que se podrían mejorar y ampliar, pero este analizador tiene características muy interesantes.

Una característica del analizador es que sienta unas bases firmes para lo que en un futuro podría ser una aplicación musical de mucha más envergadura. Si se llevasen a cabo las ampliaciones y mejoras que podrían realizarse, muchas de ellas fácilmente implementables si se diesen unas determinadas condiciones, como que en futuras versiones de JMusic muchos fallos actuales fuesen corregidos, este analizador podría convertirse en una aplicación muy completa e interesante para una gran cantidad de personas interesadas en la música. Además de ser una aplicación muy interesante, podría usarse para la formación de estudiantes y de músicos.

Además, el hecho de que sea muy modular, hace que sea muy fácilmente mantenible, y ampliable. Esto, como todos sabemos, es algo muy importante en una aplicación, y en este punto, nuestro analizador cumple perfectamente. Esta característica, hace viable que pudiera ser el principio de una aplicación mucho más práctica y potente. El uso de Java en la implementación, que no es un lenguaje difícil de dominar y que es un lenguaje orientado a objetos, facilita más todavía estas tareas.

Otra de las características de JMusic es su sencillez, lo que hace que cualquier persona pueda usar el programa. Para la correcta interpretación del análisis y de la partitura, lógicamente, hay que tener ciertos conocimientos de solfeo.

El analizador de nuestra aplicación, que es su punto fuerte, ha sido implementado de manera rigurosa. Para la implementación del analizador, así como las demás partes pero especialmente en esta, hemos buscado información sobre distintas técnicas que podríamos seguir. En ello ha colaborado activamente, como en los demás

problemas que hemos tenido, nuestro tutor Jaime, en diversas reuniones que hemos tenido a lo largo de todo el año.

Esta aplicación, en definitiva, es una aplicación sencilla pero cumple lo que promete. La parte del analizador armónico es la más elaborada, por ser la más importante de ellas.

5.2. Qué se puede mejorar

En nuestro analizador de armonía musical hay varias cosas que se pueden mejorar, como en cualquier aplicación.

La mayor parte de los problemas con los que nos hemos enfrentado han surgido por el mal funcionamiento de JMusic. Si no hubiese sido así, la aplicación hubiera contado con más funcionalidad de la que tiene.

Se podría mejorar por ejemplo, la edición de partituras, ya que, por problemas con JMusic, hemos tenido que limitar la edición de partituras a cambiar la altura de las notas. Originalmente, en JMusic, existe la posibilidad de añadir notas, borrarlas, etc. También sería interesante añadir a JMusic la posibilidad de añadir nuevas voces a una partitura, también podríamos cambiar el instrumento, etc. En definitiva, que la aplicación fuera además de un buen analizador un potente editor. Esto tendría mucho interés por ejemplo, tanto en ejercicios de análisis como en tareas de composición. Esta es una mejora que daría mucho juego y que haría del analizador armónico musical una aplicación mucho más interesante todavía, aunque para que esta mejora fuese realmente práctica, debería ir acompañada de la posibilidad de salvar ficheros midi.

Otra cosa que podría mejorarse es la interfaz. Por ejemplo, en los ficheros sin modulación, el análisis podría aparecer más cerca de la partitura y así el usuario podría visualizar simultáneamente más cosas. La interfaz además, es algo muy subjetivo y que a unos puede gustar más y a otros menos, por lo tanto, aquí las posibilidades de mejora son infinitas.

También podría mejorarse la lectura y el proceso de ficheros midi, para que leyese obras con dosillos y tresillos, ya que, la aplicación no es capaz de leer todos los ficheros midi. Lo ideal sería que JMusic hiciera bien esto, ya que está implementado, y que nosotros no hubiéramos tenido que usar ABC para hacer este proceso. Esto haría al analizador más simple, ya que ahora se construye el objeto Score a partir de ABC pista a pista y después lo volvemos a recorrer pista a a pista para generar el análisis.

El código de JMusic también admite muchísimas mejoras. Esto no es parte de nuestro proyecto, pero sería interesante optimizar el código porque podría verse muy afectado el rendimiento. Una de las mejoras más importantes que le hemos introducido al código de JMusic es la paginación, y ha incrementado notablemente el rendimiento, al tener que hacer menos trabajo por solo mostrar una parte de la partitura y del análisis y además, ha reducido muchísimo el consumo de memoria, ya que con el código original de JMusic la aplicación lanzaba excepciones de memoria. En el código de JMusic se usan métodos de Java que están cayendo en desuso y que deberían ser sustituidos por los nuevos métodos para asegurar la continuidad de la librería a medida que salen nuevas versiones de Java, pero eso es algo que corresponde a los desarrolladores de JMusic.

También es posible modificar el analizador en Prolog para que interprete acordes

usando otras reglas y heurísticas. Cada músico podría modificar el analizador a su antojo dejando intactas el resto de las partes de la aplicación y de este modo, al abrir un midi, obtendría un análisis completamente distinto más cercano a sus gustos.

El proceso de selección de pistas puede ser mejorado. En lugar de cargar el midi de nuevo cada vez que queramos cambiar las pistas seleccionadas, se podría hacer que aparezcan todas las pistas que hay, y una casilla de verificación en cada una de ellas para seleccionar las que tienen que ser consideradas y las que no, tanto para el análisis como para la reproducción. Así por ejemplo, para ver el efecto que tiene sobre el análisis eliminar o añadir una pista, bastaría con realizar la selección adecuada y darle al botón "Actualizar análisis" en lugar de tener que volver a abrir el fichero midi.

Hay que comentar también, que a medida que salgan nuevas versiones tanto de JMusic como de ABC, la aplicación mejorará su funcionamiento, al ser dependiente de las dos anteriores.

5.3. Posibles ampliaciones

Aquí vamos a comentar qué cosas son ampliables y aproximadamente cuánto esfuerzo es necesario para llevar dichas ampliaciones a cabo.

En primer lugar hay que destacar que nuestro analizador de armonía musical es muy modular, porque consta de tres partes claramente diferenciadas y separadas que interactúan entre sí para atender las peticiones del usuario. El procesador midi es un fichero .jar que usa la interfaz y el analizador en Prolog, es un fichero ejecutable que usa la interfaz cuando es necesario. Por lo tanto, son tres partes claramente diferenciadas tanto conceptualmente como "físicamente", como módulos totalmente separados que son. ¿Qué quiere decir esto?

Quiere decir, que cualquier programador con los conocimientos suficientes, podría modificar cualquier aspecto de la interfaz sin interferir en nada en la lectura de los ficheros midi ni en la forma en que el analizador lleva a cabo el análisis.

De la misma forma, también es posible ampliar o modificar el analizador sin que influya en absoluto en cualquier otro módulo de la aplicación. Se pueden modificar las reglas o heurísticas que el analizador considera para llevar a cabo el análisis y el resto de los módulos se comportarán de la misma forma que antes, siempre que se respeten los formatos que tienen que tener entradaProlog.txt y salidaProlog.txt y las interpretaciones de los mismos. Es decir, que cambiando la base de hechos del módulo en Prolog se pueden usar muchas reglas definidas para reconocer otro tipo de armonías no tonales sin variar el resto de la aplicación.

Del mismo modo, si mejoramos o ampliamos el procesamiento de midis, mientras siga creando un objeto Score que pueda tratar JMusic y los cambios respeten el formato de entradaProlog.txt, este cambio no afectará a las otras partes. Por ejemplo, podríamos hacer que el procesador de midi fuera capaz de leer ficheros midi con dosillos o tresillos, y el analizador y la interfaz seguirían funcionando exactamente de la misma forma, con lo que habríamos mejorado la aplicación con los mínimos cambios.

En cuanto a la mejora de la aplicación, comentábamos antes que una gran mejora sería hacer el editor de partituras más potente, pero esto no serviría de mucho si no somos capaces de guardar ficheros midi. Añadir la posibilidad de guardar ficheros midi, nos daría mucho juego, junto con la mejora anteriormente mencionada, porque así, el usuario sería capaz de cambiar la partitura de un fichero midi de una forma bastante detallada, podría ver los cambios producidos en el análisis debidos a dicha modificación y guardar la partitura modificada como fichero midi. Tendríamos dos maneras de añadir la posibilidad de guardar ficheros midis. La primera es esperar a que salga una versión de JMusic que solucionase el problema y, teniendo un objeto Score, sólo tendríamos que llamar al método *midi* del objeto Write de JMusic. La otra opción, mucho más compleja pero quizá más inmediata, es traducir un Score a notación ABC y usar abc2midi para guardar la información en forma de fichero midi.

Otra posible ampliación, sería pasar la secuencia de acordes obtenida por nuestra aplicación a Genaro, que era un proyecto de S.I. del curso 2004/2005 que creaba secuencias armónicas usando gramáticas regulares. Si en vez de generar gramáticas regulares, le pasamos una secuencia de acordes, obtendríamos un acompañamiento para la obra que nosotros queramos.

JMusic es capaz de guardar un objeto Score en un fichero xml y también es capaz de abrir un fichero xml y crear un objeto Score y por lo tanto, de representar su partitura. El problema es que esta característica no funcionaba siempre bien en JMusic, por lo tanto sería necesario mejorar JMusic. Si en futuras versiones de la librería funciona, sería muy interesante añadirlo a la aplicación porque en un fichero xml puede verse muy bien la información que contiene un fichero midi. Si en futuras versiones de JMusic esto está solucionado, añadirlo a nuestra aplicación sería muy fácil. Se abriría y se guardaría un fichero xml de la misma forma que un fichero midi, excepto que habría que llamar al método *xml* de los objetos Read y Write de JMusic respectivamente.

También sería muy interesante, que al reproducir sonara el instrumento indicado, y no un piano como suena ahora. Esto es otra consecuencia del uso de JMusic, puesto que está hecho de esta manera. Si en versiones posteriores se modificase esto, lo único que habría que hacer en la aplicación para que este cambio se produjese, es añadir el código fuente de JMusic a las clases de la aplicación.

Otra mejora que incorporan multitud de editores de partituras es, que cuando le damos a reproducir una partitura, tienen un marcador, que indican en todo momento qué notas son las que están sonando. Esto es muy útil para poder ir leyendo la partitura durante la reproducción y poder ver si se corresponde con lo que queremos obtener.

Capítulo 6

Conclusiones

Durante todo este curso trabajando en esta aplicación hemos aprendido muchas cosas. Es muy importante la planificación y saber en todo momento qué es exactamente lo que se quiere hacer. Esto lo hemos tenido claro durante todo el curso, pues cada uno de nosotros tenía clara cuál era su función. En ocasiones, hemos tenido que ayudarnos unos a otros para poder seguir adelante con problemas que se nos han resistido, de forma que también ha habido, como es de esperar, trabajo en equipo.

En algunos momentos hemos visto que nos aparecían demasiados problemas y cuando todo iba mejor, volvían los problemas. Pero la mayor parte de ellos han sido superados a base de esfuerzo y constancia.

Como hemos comentado antes, esta aplicación puede servir perfectamente para ser la base de otra más completa, y si no es así, al menos, merece la pena pensar que la implementación que hemos realizado, la información que hemos recopilado y las técnicas que hemos seguido les puedan servir de guía a otros alumnos que realicen proyectos relacionados.

Estamos contentos con el desarrollo y el trabajo realizado durante todo el curso. Sabíamos que era una tarea complicada, pero desde el principio estaba claro lo que había que hacer y cada uno tenía su parte de trabajo totalmente separada del trabajo del otro. La organización del trabajo, en este sentido, ha sido satisfactoria. Cuando uno de nosotros tenía una nueva versión de lo que estamos desarrollando, simplemente había que cambiar la versión anterior por la nueva en la interfaz y de la forma más sencilla, ya funcionaba la nueva versión con todo lo demás. Muy pocas veces había que realizar algún cambio en el código de la interfaz para integrar una nueva versión. Esto también ha facilitado enormemente el trabajo entre los tres, y ha sido el mayor acierto que hemos tenido durante todo el desarrollo.

También hemos aprendido que "informatizar" la música no es una tarea fácil. Solo el proceso de leer un midi, tiene mucha más complejidad de lo que puede parecer. A partir de ahí, las complicaciones se multiplican. Solo representar una partitura en un programa informático es una gran complicación, ya que no hay tantas herramientas que faciliten esto. Por ejemplo, en un principio pensamos en usar Lylipond para representar las partituras. Lylipond permite crear un fichero pdf a partir de un fichero de texto con las especificaciones de la partitura en un lenguaje parecido a Latex. El problema de esto, es que no nos permite editar la partitura y habría que mostrarla en una ventana diferente a la de la aplicación, porque habría que lanzar el lector de pdf predeterminado del sistema. Desechamos esta solución en

cuanto conocimos JMusic, porque era muy interesante la opción de que el usuario pudiera cambiar una partitura y pudiera ver los efectos de esos cambios en el análisis.

Aunque la implementación haya tenido dificultades en algunos momentos, ha merecido la pena todo el trabajo realizador, el trabajo en equipo y el resultado ha sido satisfactorio, porque hemos conseguido implementar lo que nos proponíamos.

Además, también hemos aprendido mucho en el campo de la programación. Aunque todos sabíamos programar en Java, esta aplicación ha servido para que ampliemos mucho más nuestros conocimientos sobre este lenguaje y hayamos tenido que implementar cosas con las que nunca antes nos habíamos enfrentado. Esto hace que el dominio del lenguaje sea mucho mayor. Aunque la novedad más importante ha sido el analizador en Prolog, ya que nosotros no sabíamos demasiado sobre Prolog. La posibilidad de implementar este analizador, nos ha hecho entrar de lleno en la implementación en Prolog, de la que teníamos una ligera idea, ya que el analizador es un módulo de bastante complejidad que usa muchísimas reglas y heurísticas para llevar a cabo el análisis. Es la base de nuestra aplicación.

La implementación de un sistema experto, en este caso experto en análisis armónico, no es normalmente una tarea fácil y no lo ha sido en este caso. Para llevar a cabo este objetivo, han sido determinantes la ayuda de nuestro profesor director y nuestra capacidad para buscar información por nuestra cuenta, sobre todo por Internet, ya que, los problemas que no nos han surgido por JMusic, han sido problemas conceptuales en las partes del analizador y del procesador midi, y típicos problemas de programación en los tres módulos que forman la aplicación, e Internet es la mejor herramienta para encontrar información muy específica.

Por último, hay que comentar que, si todo nos hubiera salido como esperábamos con JMusic y hubiera funcionado de forma correcta, hubiéramos tenido mucho más tiempo para mejorar la aplicación y para incluir nuevas funcionalidades. Al menos, es muy importante que, si algún día, estos problemas son solucionados en JMusic, es muy fácil adaptar la aplicación para que aproveche todas estas mejoras, así como sería bastante sencilla su ampliación.

Capítulo 7

Bibliografía

Para la elaboración de esta aplicación así como para profundizar en los contenidos tratados se pueden consultar las siguientes fuentes:

1. ROWE, R. Machine musicianship. The MIT Press, Massachusetts, 2.001.
2. CEBALLOS, Fco Javier. Java 2 Curso de programación. Rama, 2000.
3. Solomons Music Theory Resources. *http://www.solomonsmusic.net*.
4. PISTON, W. Armonía. Idea Books, Barcelona, 2.001.
5. jMusic - Computer music composition in Java. *http://jmusic.ci.qut.edu.au/*.
6. MIDI File format - *http://www.borg.com/~jglatt/tech/midifile.htm*.
7. JFugue - Java API for Music programming - *http://www.jfugue.org/javadoc/*.
8. The ABC Musical notation language - *http://staffweb.cms.gre.ac.uk/~c.walshaw/abc/*.
9. How to interpret ABC music notation - *http://www.lesession.co.uk/abc/abc_notation.htm*.
10. Tutorial sobre Swing y JFC (Java Foundation Classes) - *http://www.programacion.com/java/tutorial/swing*.